

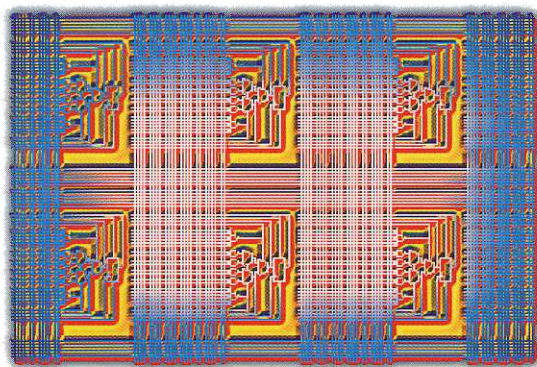
**Jacques Weber    Maurice Meaudre**

---

*IUT • 2<sup>e</sup> CYCLE • ÉCOLES D'INGÉNIEURS*

# **Le langage VHDL**

**Cours et exercices**



**2<sup>e</sup> édition**

**DUNOD**

## **Avant-propos à la seconde édition**

Depuis la première édition de cet ouvrage, en octobre 1997, l'intérêt de l'utilisation d'un langage évolué (VHDL ou Verilog) dans la modélisation et la conception des circuits intégrés n'a fait que se confirmer. Tous les fabricants de circuits logiques programmables, pour citer un exemple industriel, proposent des solutions qui font appel à ces langages. Leur introduction dans l'enseignement s'est donc révélé être un choix pertinent.

La principale nouveauté des trois dernières années concerne l'extension de ces langages vers le monde des circuits analogiques et mixtes (analogiques et numériques). Nous n'avons pour l'instant pas pris en compte ces extensions pour plusieurs raisons :

- Ces extensions concernent la modélisation des circuits, pas leur synthèse automatique (compilateurs de silicium) ; notre propos étant essentiellement les méthodes de conception des circuits numériques, l'extension analogique n'aurait fait qu'alourdir de façon importante la présentation d'un sujet déjà dense.
- Les outils informatiques (compilateurs et simulateurs) associés sont, pour l'instant, peu nombreux et coûteux, leur utilisation dans l'enseignement est loin d'être une pratique courante.

La première édition de cet ouvrage était accompagnée d'un CD-rom qui contenait les exemples du livre. En accord avec l'éditeur nous avons pensé qu'il était plus souple de renvoyer le lecteur intéressé à un site Internet :

<http://perso.wanadoo.fr/jacques.weber/>

Ce site contient les programmes sources du livre, les énoncés des exercices avec leurs corrigés et des liens vers les fournisseurs de logiciels de simulation. Les auteurs tiennent à remercier ici les étudiants de l'IUT de Cachan qui contribuent, au cours de travaux de projets tutorés à alimenter le contenu de ce site.

# Table des matières

<b>AVANT-PROPOS À LA SECONDE ÉDITION</b>	<b>V</b>
<b>AVANT-PROPOS</b>	<b>1</b>
<b>CHAPITRE 1 • MODÉLISATION ET SYNTHÈSE : LE MÊME LANGAGE</b>	<b>5</b>
1.1 Simulation et synthèse	6
1.1.1 Un langage commun	6
1.1.2 Tout n'est pas synthétisable	7
1.1.3 Simulation fonctionnelle	8
1.1.4 Du langage au circuit : la synthèse	8
1.1.5 Du circuit au langage : la modélisation	10
1.2 Portabilité	11
1.2.1 Indépendance vis-à-vis du circuit cible	11
1.2.2 Indépendance vis-à-vis du compilateur	11
1.3 Un langage puissant	12
1.3.1 Construction hiérarchique	12
1.3.2 Description fonctionnelle	13
<b>CHAPITRE 2 • VHDL</b>	<b>15</b>
2.1 Un préliminaire grammatical : le formalisme de Backus et Naur (BNF)	16
2.2 Premier aperçu	17
2.2.1 L'exemple incontournable : une commande de feux tricolores	18
a) <i>Ce que l'on veut obtenir</i>	18
b) <i>La façon de le réaliser</i>	21

2.2.2	Le couple entity architecture	24
a)	<i>La boîte noire : une entité</i>	24
b)	<i>Son contenu : une architecture</i>	27
2.2.3	Types et classes	30
a)	<i>Les types scalaires</i>	32
b)	<i>Les types structurés</i>	34
c)	<i>Pointeurs et fichiers</i>	35
d)	<i>Constantes signaux et variables</i>	36
e)	<i>Alias</i>	40
2.2.4	Expressions	41
a)	<i>Opérateurs</i>	41
b)	<i>Opérandes</i>	43
2.2.5	Attributs	46
a)	<i>Attributs prédéfinis</i>	47
b)	<i>Attributs définis par l'utilisateur</i>	51
2.3	Parallélisme et algorithmes séquentiels	51
2.3.1	Le corps d'une architecture : le monde concurrent	53
a)	<i>Affectations de signaux</i>	53
b)	<i>Instanciation de composants</i>	59
c)	<i>Modules de programme</i>	61
2.3.2	Le corps d'un processus : le monde séquentiel	66
a)	<i>Une boucle sans fin contrôlée par des événements</i>	67
b)	<i>Signaux et variables</i>	68
c)	<i>Instructions séquentielles</i>	71
2.3.3	Modélisation des opérateurs logiques combinatoires	75
a)	<i>Par des instructions concurrentes</i>	75
b)	<i>Par un algorithme séquentiel</i>	77
2.3.4	Modélisation des opérateurs séquentiels	78
a)	<i>Opérateurs synchrones et asynchrones</i>	79
b)	<i>Machines d'états synchrones</i>	83
2.4	Quelques pièges	91
2.4.1	Les mémoires cachées	92
a)	<i>Un compteur à maintien asynchrone</i>	92
b)	<i>Les indices de l'anomalie</i>	93
c)	<i>Les remèdes</i>	94
2.4.2	Signaux et variables	96
a)	<i>Un générateur de parité fantaisiste</i>	96
b)	<i>Variables et bascules</i>	97
2.4.3	Les boucles	99
a)	<i>Des boucles non synthétisables</i>	99
b)	<i>Des boucles inutiles</i>	100
2.4.4	La complexité sous-jacente	101
a)	<i>Les opérations arithmétiques ou l'explosion combinatoire</i>	102
b)	<i>Les horloges multiples</i>	103



2.5	Programmation modulaire	104
2.5.1	Les sous-programmes : procédures et fonctions	104
a)	<i>Syntaxe</i>	105
b)	<i>Surcharges de sous-programmes</i>	110
2.5.2	Librairies et paquetages	112
a)	<i>Fichiers sources et unités de conception</i>	113
b)	<i>La librairie work</i>	116
c)	<i>La librairie std</i>	116
d)	<i>La librairie IEEE et la portabilité des sources</i>	117
2.6	Construction hiérarchique	119
2.6.1	Blocs externes et blocs internes	120
a)	<i>Localité des noms</i>	120
b)	<i>Paramètres génériques</i>	120
2.6.2	L'instruction generate	122
a)	<i>Boucles et tests dans l'univers concurrent</i>	122
b)	<i>Instanciations multiples : des schémas algorithmiques</i>	123
2.6.3	Configuration d'un projet	124
a)	<i>Spécification de la configuration d'un composant</i>	125
b)	<i>Déclaration de configuration d'une entité</i>	125
2.7	Modélisation et synthèse	128
2.7.1	Tout ce qui est synthétisable doit être simulable	128
2.7.2	La réciproque n'est pas vraie mais ...	128
a)	<i>Penser « circuit »</i>	131
b)	<i>De l'ordre dans les horloges</i>	131
2.7.3	Pilotes multiples et fonctions de résolution	132
a)	<i>Conflits</i>	132
b)	<i>Sorties non standard (trois états et collecteurs ouverts)</i>	135
c)	<i>Signaux gardés</i>	136
2.7.4	Les paquetages de la librairie IEEE	136
a)	<i>Des types logiques simples multivalués</i>	136
b)	<i>Des opérations prédéfinies</i>	140
2.7.5	Des outils de simulation	141
a)	<i>Les fichiers</i>	142
b)	<i>Le traitement des fautes</i>	148
c)	<i>La gestion du temps simulateur</i>	150
d)	<i>Les retards dans les circuits</i>	151
2.7.6	Rétroannotation	156
a)	<i>Les modèles postsynthèse</i>	158
b)	<i>Vital</i>	159
2.7.7	Construire un « test bench »	165
a)	<i>Attention au mélange de genres</i>	165
b)	<i>Une boîte à outils</i>	166
c)	<i>Une configuration générale de test</i>	173
d)	<i>Simulation en boucle fermée</i>	176
c)	<i>En matière de conclusion</i>	186

<b>CHAPITRE 3 • LES CIRCUITS PROGRAMMABLES</b>	<b>187</b>
3.1 Les grandes familles	188
3.2 Qu'est-ce qu'un circuit programmable ?	190
3.2.1 Des opérateurs génériques	190
a) Réseaux logiques programmables	190
b) Multiplexeurs	192
c) Ou exclusif	194
d) Bascules	195
3.2.2 Des technologies	195
a) Fusibles	195
b) MOS à grille flottante	197
c) Mémoires statiques	198
d) Antifusibles	199
3.2.2 Des architectures	199
a) Somme de produits	199
b) Cellules universelles interconnectées	200
c) Cellules d'entrée-sortie	201
e) Placement et routage	202
3.2.4 Des techniques de programmation	202
a) Trois modes : fonctionnement normal, programmation et test	203
b) Programmables in situ	204
3.3 PLDs, CPLDs, FPGAs : Quel circuit choisir ?	205
3.3.1 Critères de performances	205
a) Puissance de calcul	205
b) Vitesse de fonctionnement	207
c) Consommation	208
d) L'organisation PREP	209
3.3.2 Le rôle du « fitter »	211
a) « Mise à plat » ou logique multicouches ?	211
b) Surface ou vitesse	212
c) Librairies de macrofonctions	212
d) Le meilleur des compilateurs ne peut donner que ce que le circuit possède	212
<b>EXERCICES</b>	<b>213</b>
<b>Annexes</b>	<b>220</b>
<b>BIBLIOGRAPHIE</b>	<b>226</b>
<b>INDEX</b>	<b>228</b>

# Avant-propos

Au cours des vingt dernières années, les méthodes de conception des fonctions numériques ont subi une évolution importante. Dans les années soixante-dix, la majorité des applications de la logique câblée étaient construites autour de circuits intégrés standard, souvent pris dans la famille TTL. Au début des années quatre-vingt, appaurent, parallèlement, les premiers circuits programmables par l'utilisateur, du côté des circuits simples, et les circuits intégrés spécifiques (ASICs), pour les fonctions complexes fabriquées en grande série. La complexité de ces derniers a nécessité la création d'outils logiciels de haut niveau, qui sont à la description structurale (schémas au niveau des portes élémentaires) ce que les langages évolués sont au langage machine dans le domaine de la programmation. À l'heure actuelle, l'écart de complexité entre circuits programmables et ASICs s'est restreint : on trouve une gamme continue de circuits qui vont des héritiers des premiers PALs (*programmable array logic*), équivalents de quelques centaines de portes, à des FPGAs (*Field programmable gate array*) ou des LCAs (*Logic cell array*) de plus de cent mille portes équivalentes. Les outils d'aide à la conception se sont unifiés ; un même langage, VHDL par exemple, peut être employé, quels que soient les circuits utilisés des PALs aux ASICs.

*VHDL du langage au circuit, du circuit au langage* est l'un des résultats de la réflexion menée à l'IUT de Cachan : quelle évolution imprimer au contenu de l'enseignement de l'électronique numérique, pour proposer aux étudiants une formation en accord avec les méthodes de conceptions actuelles ?

Nous associons étroitement l'apprentissage des bases de l'électronique et celui des méthodes modernes de conception. VHDL est vu, dans cette optique, comme un outil, abordé en annexe du cours d'électronique numérique. L'intérêt de cette approche est son caractère démystificateur : ce langage, réputé comme complexe, est apprivoisé à travers les résultats du processus de synthèse. Une approche ascendante,

en quelque sorte. Au niveau IUT il est exclu d'étudier tous les aspects du langage. L'important volet de la modélisation est donc, à ce niveau, délibérément omis.

Notre approche a, malgré ses restrictions volontaires, rencontré un accueil favorable dans les milieux professionnels de la conception de circuits et dans des formations universitaires ou d'ingénieurs.

L'ouvrage présenté ici est l'aboutissement logique de notre démarche : les aspects de synthèse sont omniprésents, complétés par l'aspect modélisation. Le lecteur découvrira donc le chemin qui va du modèle fonctionnel au circuit, mais aussi (une sorte de retour aux sources) celui qui va du circuit à son modèle (rétroannoté), défauts et limitations compris. La mise en relation de ces différentes visions que l'on a d'un circuit numérique nécessite la création de bancs de tests, *test benches* pour les initiés. Nous présentons au lecteur une méthodologie générale de création de ces programmes de test qui font partie intégrante de tout projet complexe.

Le monde VHDL s'est fortement unifié au cours des cinq dernières années (1995-2000) ; les éparpillements des débuts des outils de synthèse sont en passe d'être oubliés. Les librairies standardisées IEEE fournissent un cadre homogène et portable pour la synthèse et la modélisation. Les contenus de ces librairies sont explicités, y compris celui des modèles VITAL.

Le livre est subdivisé en trois parties, de volumes très inégaux :

- Une introduction générale aux concepts utilisés dans la suite. On ne découvrira rien, ou presque, du langage dans cette partie ; mais on préparera le cadre intellectuel dans lequel la description du langage lui-même va s'insérer.
- Le langage lui-même. De cette partie, la principale du livre, doit ressortir une vision que nous espérons suffisamment complète et explicite pour que le lecteur puisse se jeter à l'eau. Une grande importance est accordée ici aux méthodes de conception modulaire et au test. Si le lecteur en a l'opportunité, le cas ne devrait pas être rare compte tenu du sujet, il est chaleureusement convié à explorer les sources des librairies (IEEE, VITAL) fournies avec l'outil de développement dont il dispose. Elles représentent une mine d'informations dont nous nous sommes amplement servis.
- Un guide de lecture des notices de circuits programmables. Nous avons volontairement rejeté l'approche catalogue, qui serait forcément obsolète dès la publication de l'ouvrage, pour dégager les lignes directrices et aider le lecteur à décrypter des documentations pour initiés (que veut dire PREP, par exemple). Au-delà des valeurs chiffrées en nombre de portes et mégahertz, les familles de circuits obéissent à des philosophies de conception et à des contraintes technologiques. Ici également, l'accès à une documentation constructeur sera un complément de lecture utile.

De nombreux exemples illustrent les principes abordés, et des exercices permettent au lecteur d'asseoir sa compréhension.

La connaissance d'un langage de programmation procédural, comme C ou Pascal, n'est pas indispensable, mais elle facilite grandement la compréhension de certains

passages. Des connaissances de base sur les circuits numériques et leurs méthodes de conception, au niveau architectural pour le moins, sont plus que souhaitables.

Nos remerciements vont aux étudiants, les premiers concernés ; aux auditeurs de formations qui nous ont, par leurs encouragements (qui allaient jusqu'à en redemander) récompensés de notre effort et, par leurs critiques minutieuses, aidés à cerner les directions à suivre ; à nos collègues, bien sûr, premiers cobayes de ce texte à qui nous devons bien des approfondissements.

## Chapitre 1

---

# Modélisation et synthèse : le même langage

Représentation spatiale et analyse séquentielle<sup>1</sup> : la première privilégie le global, l'organisation, les relations, dans un traitement parallèle (et/ou simultané); la seconde décompose, analyse les successions d'événements, décrit les causalités.

Les deux visions sont nécessaires à la description des circuits électroniques. Elles produisent, dans les approches traditionnelles, deux catégories de langages, complémentaires mais disjoints : les schémas et les algorithmes<sup>2</sup>. Tenter de saisir le fonctionnement d'un Pentium à partir de son schéma électrique est illusoire, l'expliquer par un algorithme également<sup>3</sup>.

Les langages de description du matériel, traduction littérale de *hardware description language* (HDL), se doivent de marier les deux approches. Le même langage permet, à tous les niveaux d'une représentation hiérarchique, d'associer les aspects structurels et les aspects fonctionnels, l'espace et le temps. VHDL en est un, il y en a d'autres, VERILOG, par exemple. Nous avons choisi, ici, d'explorer un peu le premier.

- 
1. Hémisphères droit et gauche du cerveau humain, diront certains. Mais c'est une autre histoire.
  2. Peu importe, à ce niveau, le support physique utilisé, graphique ou textuel. Un schéma peut être décrit par un dessin, le monde graphique, ou par une *net list*, le monde textuel. De même, un algorithme peut être décrit par un diagramme (*flowchart*), le monde graphique, ou par un langage procédural, le monde textuel. Chaque représentation a ses avantages et ses inconvénients.
  3. La deuxième tentative est, en outre, fausse. Elle ne prend pas en compte le parallélisme inhérent à tout système électronique.

L'origine du langage suit une histoire un peu banale dans sa répétition<sup>1</sup> : le développement de l'informatique, la complexité croissante des circuits (numériques pour l'instant), ont fait fleurir une multitude de langages plus ou moins compatibles entre eux, plus ou moins cohérents entre synthèse (fabrication) et simulation (modélisation). Le programme VHSIC (*very high speed integrated circuits*), impulsé par le département de la défense des États-Unis dans les années 1970-1980, a donné naissance à un langage : VHSIC-HDL, ou VHDL.

Deux normes successives (IEEE<sup>2</sup>-1076-1987 et 1993) en ont stabilisé la définition, complétées par des extensions plus récentes ou à venir, nous en verrons certaines. L'évolution prévue est la création d'un langage commun analogique-numérique, dont VHDL, sous sa forme actuelle, constituerait la facette numérique. Certains outils de CAO ont déjà entamé leur évolution dans cette voie.

« Apprendre à communiquer en une langue nouvelle prend toujours l'allure d'un défi. Au début, il semble que tout est nouveau, et doit être assimilé avant de caresser l'espoir d'utiliser le langage. Puis, après une première expérience, il apparaît des points de ressemblance avec notre propre langage. Au bout d'un moment, nous réalisons que les deux langages ont de nombreuses racines communes. »

Cette première phrase de *ADA une introduction*, par H.Ledgard, résume on ne peut mieux la difficulté. Par où commencer ?

Nous commencerons par les principes, les idées directrices, avant de nous jeter à l'eau dans le chapitre suivant.

## 1.1 SIMULATION ET SYNTHÈSE

Les différentes étapes de la conception d'un circuit intégré sont illustrées par le diagramme de la figure 1-1.

L'ensemble du processus qui permet de passer de la vision fonctionnelle au circuit constitue la synthèse.

### 1.1.1 Un langage commun

Le principe majeur de VHDL est de fournir un langage commun, depuis la description au niveau système jusqu'au circuit. À tous les niveaux de description le langage est le même.

Le découpage en étapes, illustré par la figure 1-1, n'est pas toujours aussi schématique, il peut y en avoir plus comme il peut en manquer. En particulier, beaucoup de logiciels de synthèse de circuits programmables quittent le monde VHDL pour les

---

1. Toute ressemblance avec ADA serait purement fortuite.

2. *Institute of Electrical and Electronics Engineers.*

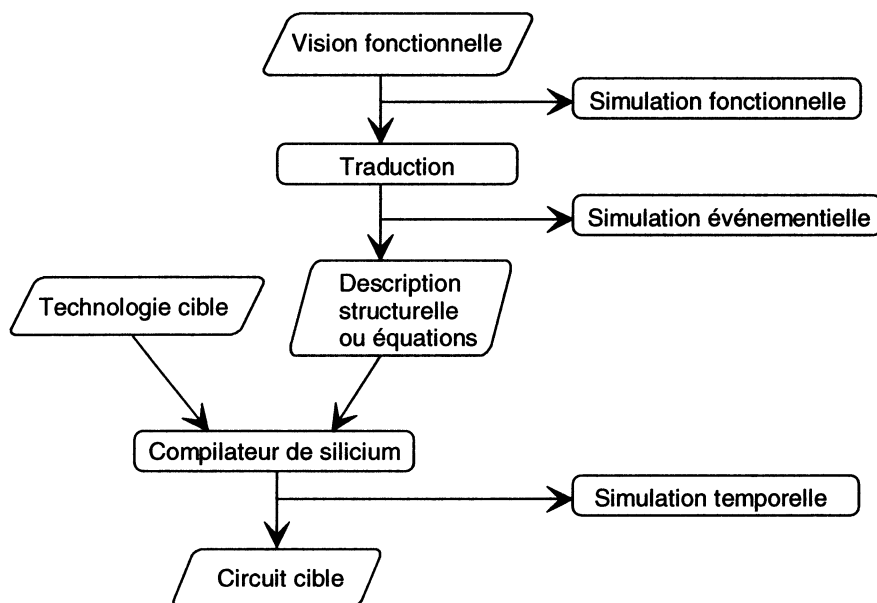


Figure 1-1 Étapes de conception d'un circuit.

étapes intermédiaires<sup>1</sup>. Ils le retrouvent, heureusement, à la dernière étape qui permet la validation ultime de la conception avant la réalisation matérielle du circuit.

### 1.1.2 Tout n'est pas synthétisable

Créer un langage, qui permette à la fois de valider une conception au niveau système, de construire les programmes de test utiles à tous les niveaux et de générer automatiquement des schémas d'implantation sur le silicium, est évidemment un propos ambitieux. VHDL est donc beaucoup plus qu'une simple traduction textuelle de la structure interne d'un circuit.

Les constructions légales ne sont pas toutes synthétisables, seul un sous-ensemble des instructions permises par le langage ont un sens pour la synthèse. L'aboutissement, heureux cela va sans dire, d'un projet implique une vision précise de la différence entre la réalité du silicium et le monde virtuel qui sert à le concevoir. Comme tout langage de haut niveau, VHDL est modulaire. Cette modularité permet de voir un schéma complexe comme l'assemblage d'unités plus petites, elle contribue également à séparer le réel du virtuel.

Sans rentrer ici dans des exemples techniques, nous les aborderons plus avant, indiquons qu'un modèle VHDL peut détecter des incohérences dans les signaux qui

1. Il y a là un problème de portabilité « latéral », les outils de conception de circuits existent évidemment depuis plus longtemps que VHDL. Ces outils ont leurs langages, parfois liés à des fabricants ; le traducteur joue alors le rôle de passerelle vers ces langages.



lui sont transmis, des fautes de *timing*, par exemple. Face à une telle situation le modèle peut interagir avec le simulateur, modifier les conditions du test, avertir l'utilisateur par des messages ou consigner des résultats dans un fichier. De tels modules de programme ne sont évidemment pas synthétisables. Parfois la distance entre synthèse et modélisation est plus ténue : générer un retard de 13 nanosecondes, par exemple, n'est pas, dans l'absolu impensable. Mais comment créer ce retard dans un circuit cadencé par une horloge à 100 MHz ?

### 1.1.3 Simulation fonctionnelle

Tester un circuit revient à lui appliquer des stimuli, des vecteurs de test, et à analyser sa réponse. À un deuxième niveau, le test inclut l'environnement dans lequel le circuit est immergé, la carte sur laquelle il est implanté. Les conditions expérimentales reproduiront, dans un premier temps, les conditions de fonctionnement normales ; dans un deuxième temps elles chercheront à cerner les limites, une fréquence maximum, par exemple.

Tester un module logiciel revient à écrire un programme qui lui transmet des données, reçoit les résultats et les analyse. Dans une première étape les données transmises sont celles attendues, elles servent à identifier d'éventuelles erreurs grossières ; dans une deuxième étape, les données sortent de l'ordinaire prévu, provoquent une situation de faute. Les exemples informatiques abondent, en voici un : quand on réalise un programme de tracé de courbes, une calculatrice graphique, demander le tracé de  $y = x^2$  est un test de la première catégorie, celui de  $y = \text{Log}[\text{Sin}(1/x^2)]$  de la seconde<sup>1</sup>.

VHDL est un langage informatique qui décrit des circuits. Les programmes de test simulent l'environnement d'un montage, ils bénéficient de la souplesse de l'informatique pour recréer virtuellement toutes les situations expérimentales possibles et imaginables, même celles qui seraient difficiles à réaliser en pratique.

Le test est une opération psychologique difficile à mener : nous avons une fâcheuse tendance à vouloir nous convaincre nous-mêmes de la justesse de notre réalisation, à construire des tests qui nous conforteront dans cette idée. Or le but du test est de chercher la faute, même, et surtout, la faute maligne, celle qui ne se révèle pas au premier abord<sup>2</sup>.

### 1.1.4 Du langage au circuit : la synthèse

Le synthétiseur interprète un programme et en déduit une structure de schéma. Dans une deuxième étape, il cherche à reproduire ce schéma dans une technologie, au moindre coût (surface de silicium), avec l'efficacité la plus grande (vitesse maximum).

---

1. Dans une première étape on peut supprimer le logarithme. L'intervalle de variation intéressant est évidemment centré autour de 0, comme  $\{-1..1\}$ .

2. Une situation de débordement dans un calcul numérique, par exemple.

Considérons un exemple simple : on souhaite réaliser la somme de deux nombres entiers, et stocker le résultat dans un registre synchrone, qui dispose d'une remise à zéro synchrone.

La partie active du code VHDL, traduisant ces opérations élémentaires, est indiquée ci-dessous :

```

somme : process -- ceci est un commentaire,
              -- il va jusqu'à la fin de la ligne.
begin
    wait until hor ='1' ; -- attente du front montant
    if raz = '1' then      -- commande de remise à zéro
        reg <= 0 ;
    else                   -- addition
        reg <= e1 + e2 ;
    end if ;
end process somme ;

```

De ce programme le synthétiseur infère la présence d'un registre synchrone (des bascules D) pour stocker le résultat reg, grâce à l'instruction d'attente d'un front d'horloge. Il infère un bloc de calcul qui renvoie la somme des deux opérandes et l'affecte au contenu du registre.

D'où la structure de la figure I-2.

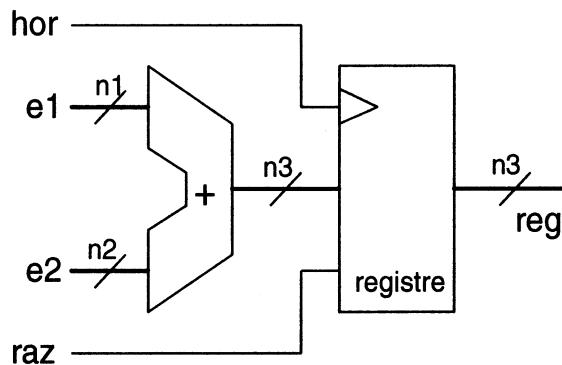


Figure 1.2 Un additionneur.

Dans cette architecture, de nombreux points restent à préciser : quels sont les types des opérandes et du résultat ? Quelles sont leurs tailles, en nombre de chiffres binaires ? Comment réaliser l'addition compte tenu du circuit cible ?

Les réponses aux premières de ces questions se trouvent dans des zones déclaratives :

```

port (hor, raz : in bit ; -- signaux binaires
      e1,e2 : in integer range 0 to 15 ; -- entiers 4 bits
      reg : out integer range 0 to 31 ) ; -- 5 bits

```

Le schéma comportera donc cinq bascules. À propos de ces premières instructions VHDL, nous découvrons l'importance des déclarations, l'omission de la restriction du domaine de variation des nombres conduirait, par défaut, à générer systématiquement un additionneur 32 bits.

La réponse à la dernière des questions précédentes ne se trouve pas dans le programme source. Elle fait partie intégrante de l'outil de synthèse, et dépend du circuit cible. Si ce dernier ne comporte comme opérateurs combinatoires élémentaires que des portes logiques, le synthétiseur devra être capable de traduire l'addition binaire en équations logiques, chiffre binaire par chiffre binaire. Si le circuit cible comporte des blocs arithmétiques, le synthétiseur tentera de les utiliser. Souvent la réalité se situe à mi-chemin entre logiciel et matériel : le circuit ne comporte que des opérateurs logiques, mais le compilateur de silicium possède des bibliothèques fournies de fonctions standard précompilées et optimisées. La traduction<sup>1</sup> du code source revient alors à reconnaître lesquelles de ces fonctions peuvent être utiles.

Le programmeur attendrait, d'un outil de CAO idéal, que toutes les réponses non inscrites dans le programme source soient fournies automatiquement, au mieux de ses intérêts, sans qu'il ait à s'en préoccuper. Ne nous leurrions pas, une bonne connaissance des circuits utilisés reste nécessaire, quelle que soit la qualité des logiciels employés.

### 1.1.5 Du circuit au langage : la modélisation

La réalité du monde physique introduit des différences de comportement entre le modèle fonctionnel et sa réalisation dans un circuit. Les principales de ces différences proviennent des temps de calcul des opérateurs. Une fois le programme compilé, traduit en un schéma de portes et de bascules interconnectées, il est intéressant de prévoir, compte tenu de la technologie employée, les limites de son fonctionnement. Cette opération de modélisation postsynthèse est connue sous le terme de *rétroannotation*.

Un modèle rétroannoté réalise, c'est du moins souhaitable, la même fonction que le modèle synthétisable dont il est issu, mais il apporte des renseignements complémentaires concernant les limites de fonctionnement que l'on peut attendre du circuit. On peut remarquer que le modèle rétroannoté n'est, lui, pas synthétisable ! Il représente une réalité qu'il est incapable de créer<sup>2</sup>.

---

1. Avec un certain humour, les auteurs d'un tel traducteur, qui s'adapte aux familles de circuits cibles, l'ont baptisé *metamor*.

2. Le rêve de tout électronicien : diminuer les temps de retard du modèle rétroannoté pour augmenter la fréquence de travail du circuit réel.

## 1.2 PORTABILITÉ

La *portabilité*<sup>1</sup> est le maître mot. Dans le monde des circuits ce mot a un double sens : portabilité vis-à-vis des circuits et portabilité vis-à-vis des logiciels de conception.

### 1.2.1 Indépendance vis-à-vis du circuit cible

Le même langage peut servir à programmer un circuit de quelques centaines de portes équivalentes, et à concevoir un circuit intégré spécifique qui en contient plusieurs millions. Les compilateurs utilisés n'ont, bien sûr, pas grand-chose de commun, le nombre de personnes impliquées dans un projet non plus ; mais le langage est le même<sup>2</sup>. Cette indépendance vis-à-vis du circuit cible a des applications industrielles : elle permet, par exemple, de commencer une réalisation avec des circuits programmables, dont le cycle de réalisation est simple et rapide, pour passer à un circuit spécifique dans une deuxième étape. La première version sert de prototype à la seconde.

### 1.2.2 Indépendance vis-à-vis du compilateur

Le deuxième aspect de la portabilité concerne les outils de développement. Le même programme est accepté par des logiciels de CAO très divers.

Historiquement, cette portabilité était excellente en simulation, c'est la moindre des choses, mais présentait des limitations sévères en synthèse. Pour un synthétiseur, certaines constructions sont un peu « magiques » : elles orientent le compilateur vers des opérations spécifiques, grâce à un sens caché qu'il est le seul à connaître. Ces particularismes locaux sont souvent liés à des définitions de types de données<sup>3</sup>, qui figurent dans des bibliothèques spécifiques d'un outil. La règle en VHDL est de fournir les codes sources des bibliothèques, règle respectée par tous les fournisseurs ; mais cette (bonne) habitude, si elle assure une portabilité sans faille des outils de synthèse aux outils de simulation, ne peut évidemment pas assurer le portage des aspects magiques

1. Nous pouvons compléter, à la lumière de ce paragraphe, une remarque précédente que nous avons faite à propos des représentations graphique et textuelle. La représentation graphique a des aspects séduisants de lisibilité, c'est évident. Elle pose cependant un problème non trivial de portabilité (rares sont les outils graphiques universels) et de rigueur. Une représentation textuelle est moins intuitive, mais ne pose pas de problème de portabilité et permet une formalisation plus aisée, qui lui assure un caractère non ambigu. De nombreux systèmes proposent les deux modes ; le mode graphique reste local, le passage d'un système à un autre se fait par l'intermédiaire du langage textuel.
2. Un langage n'est qu'un outil. Mais les langages de haut niveau ne sont pas neutres au niveau des méthodes de travail qu'ils induisent. Dans un contexte d'apprentissage, il est intéressant de disposer d'outils de développement petits et bon marché, qui permettent de développer des applications suffisamment simples pour être menées à terme par une seule personne en quelques heures.
3. Le seul type que nous avons rencontré, pour l'instant, est le type entier ; il y en a d'autres.

cachés. L'exemple caractéristique de données à caractère caché est la modélisation des bus. Le langage n'avait pas, volonté d'ouverture ou omission, prévu les logiques *trois états* dans ses types de base. En simulation rien n'est plus simple que de construire de nouveaux types, tout est prévu. En synthèse, l'apparition d'un signal qui peut prendre deux états logiques et un état haute impédance, provoque la génération d'opérateurs très particuliers, qui échappent au monde des équations booléennes. Il y a un sens caché attaché à un type trois états. Tous les fournisseurs de logiciel ont défini leurs propres librairies synthétisables, toutes sensiblement équivalentes, mais strictement incompatibles.

Un effort de standardisation a été entrepris, entre 1993 et 1995, pour assurer une portabilité des programmes tant en synthèse qu'en simulation. Entre 1995 et 1997, tous les outils de conception de circuits programmables, par exemple, ont rejoint les nouveaux standards (IEEE-1076.3).

Le même type d'évolution a eu lieu dans le domaine de la rétroannotation. Des modèles maison, portables puisque fournis sous forme de sources, mais au prix d'échanges de librairies volumineuses, on est passé, avec le standard IEEE-1076.4 (ou VITAL-95), à une compatibilité entre les bibliothèques des différents fondeurs. Cette compatibilité assure, de plus, une passerelle avec le monde VERILOG, riche en compétences et en expérience dans ce domaine<sup>1</sup>.

## 1.3 UN LANGAGE PUISSANT

Représentation spatiale et analyse séquentielle : VHDL favorise une approche hiérarchique pour la description d'un circuit complexe. Du niveau le plus élémentaire (les portes) au niveau le plus élevé (le système) le programmeur peut, définir à sa guise des sous-ensembles en relations, décrire un sous-ensemble par une représentation comportementale (analytique) ou par une nouvelle hiérarchie locale. Les deux approches peuvent coexister dans un même programme source, et l'utilisateur peut spécifier à tout moment quelle configuration il choisit d'analyser.

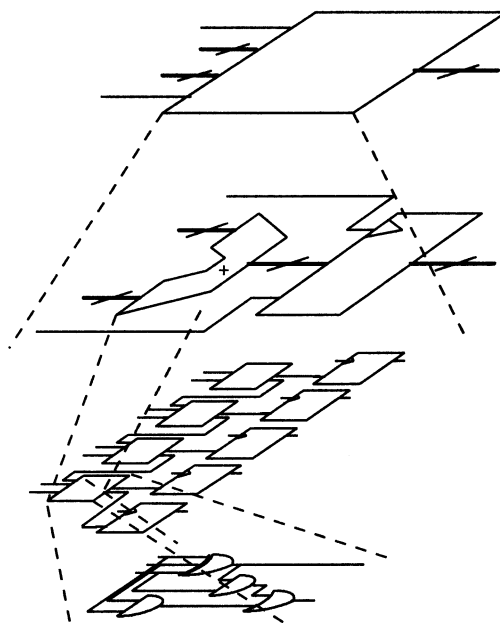
### 1.3.1 Construction hiérarchique

Reprenons l'exemple de notre additionneur. Un lecteur pointilleux pourra nous objecter que nous en avons fait une représentation graphique structurée, et que nous l'avons décrit par un programme comportemental. C'est on ne peut plus exact. Il était parfaitement possible de créer un programme VHDL qui reproduise la structure; compte tenu de la simplicité du problème, nous avons considéré que l'approche analytique était plus lisible.

La vision hiérarchique du même additionneur consiste à traiter les blocs, l'additionneur et le registre, comme assemblage d'objets plus élémentaires. La figure 1-3 en donne une image.

---

1. Une partie du paragraphe 2-7, est consacrée à ces points.



**Figure 1-3** Vision hiérarchique de l'additionneur

La construction hiérarchique n'est, bien sûr, pas propre à VHDL, tous les systèmes de CAO dignes de ce nom la comprennent.

Nous ne donnerons pas ici la façon de créer une hiérarchie dans le langage, chaque chose en son temps. Contentons-nous d'insister sur l'importance du concept et sur le fait que les instructions correspondantes sont extrêmement puissantes ; beaucoup plus qu'il n'est possible de le figurer avec des images. VHDL inclut toute une algorithmique spatiale, qui permet de créer des structures paramétrables et évolutives.

### 1.3.2 Description fonctionnelle

Complémentaire de la précédente, la vision fonctionnelle apporte la puissance des langages de programmation. Au premier abord elle est assez classique, le programmeur habitué aux langages procéduraux se retrouve en terrain connu.

Le point important, qui rend VHDL très différent de langages comme C ou PASCAL, malgré les ressemblances syntaxiques fortes avec ce dernier, est que tout algorithme est la description interne d'un bloc situé quelque part dans la hiérarchie du schéma complet.

La vision structurelle est concurrente, la vision algorithmique est séquentielle, au sens informatique du terme. Un programme VHDL doit être compris comme l'assemblage, en parallèle, de tâches indépendantes qui s'exécutent concurremment. Les signaux sont, en réalité, les véhicules des informations électriques dans le

schéma ; vus sous l'angle quelque peu informatique précédent, ils jouent le rôle de canaux de communications interprocessus.

Avant de rentrer dans le vif du sujet, autorisons-nous un conseil : n'oubliez jamais que vous décrivez un circuit. Les langages concurrents sont parfois déconcertants pour les novices, nous avons ici la chance de traduire dans un tel langage le monde des circuits ; la vision physique des choses permet de franchir sans difficulté bien des écueils.

## Chapitre 2

---

# VHDL

Comme son nom l'indique, VHDL est un langage i.e. un système de signes permettant la communication. La compréhension d'une construction (un programme) écrite dans un langage, passe par la connaissance de la grammaire qui régit l'organisation des phrases : la syntaxe, et la connaissance du sens de chaque construction élémentaire : la sémantique.

La syntaxe s'axiomatise aisément. Le manuel de référence de VHDL et de nombreuses documentations de compilateurs utilisent pour la décrire le formalisme de Backus et Naur<sup>1</sup>, *BNF* pour *Backus Naur form*. Nous commencerons par présenter rapidement ce formalisme qui, avec un peu d'habitude, se révèle très pratique. De façon à accoutumer le lecteur à son utilisation, nous utiliserons le formalisme BNF pour préciser la syntaxe de chaque construction rencontrée.

La sémantique requiert des explications informelles, des exemples, un travail de maturation beaucoup plus difficile. La suite de ce chapitre est consacrée à une découverte, forcément partielle, du sens du langage.

---

1. Pour étudier les constructions syntaxiques des langages courants, comme l'anglais, Noam Chomsky a introduit dans les années 1950 la notion de grammaire non contextuelle. Backus et Naur ont adapté les travaux de Chomsky à la description des langages informatiques (ALGOL, 1962).

Les lointaines retombées des travaux de Chomsky, dans les livres de français de nos écoles élémentaires, ont sans doute eu un effet discutable sur la maîtrise de la grammaire qu'ont acquise les collégiens des années 1980, mais ceci est une autre histoire.



## 2.1 UN PRÉLIMINAIRE GRAMMATICAL : LE FORMALISME DE BACKUS ET NAUR (BNF)

Prenons l'exemple de la description d'une expression arithmétique simple. Nous limiterons le nombre d'opérateurs connus d'un langage hypothétique à deux opérateurs additifs, + et –, et à deux opérateurs multiplicatifs, \* et /. Les opérateurs multiplicatifs ont une priorité supérieure aux opérateurs additifs, comme il se doit<sup>1</sup>. Le signe d'un objet peut être précisé ou modifié, mais ce n'est pas obligatoire, par les opérateurs unaires (à un seul opérande) + et –. Bien que les symboles soient les mêmes, les opérateurs unaires correspondent à des opérations très différentes de celles engendrées par leurs homonymes binaires (à deux opérandes); ils n'appartiennent pas à la même classe syntaxique.

Des parenthèses, ( et ), permettent de modifier l'ordre d'évaluation d'une expression.

Nous admettrons que les opérandes ne peuvent être que des nombres ou des noms (chaînes de caractères alphabétiques) de variables simples.

Par exemple :

```

325
machin
- 3 + 4 * (toto + 25)

```

sont des expressions grammaticalement correctes.

Par contre :

```

- 3 + / 4 * (toto + 25)

```

est incorrecte à cause de la succession des symboles + et /.

Comment exprimer la validité d'une expression ? Il suffit de la « raconter » :

- Une expression est formée d'un signe optionnel suivi d'un terme, suivi lui-même un nombre arbitraire de fois (éventuellement zéro fois) de doublets constitués par un opérateur additif suivi d'un terme.
- Un terme est formé d'un facteur suivi lui-même d'un nombre arbitraire (qui peut être nul) de doublets constitués par un opérateur multiplicatif suivi d'un facteur.
- Un facteur est un nombre, ou un nom, ou une expression entre parenthèses.

La traduction en « BNF » de ces règles grammaticales peut être trouvée dans le manuel de référence VHDL, dont nous nous sommes inspirés en limitant considérablement le nombre de règles, donc la richesse des expressions admises :

```

simple_expression ::= [sign] term {adding_operator term}
sign ::= + | -
adding_operator ::= + | -
term ::= factor {multiplying_operator factor}
multiplying_operator ::= * | /
factor ::= number | name | ( simple_expression )

```

1. Cela signifie que  $2 + 3 * 4$  se calcule comme  $2 + (3 * 4)$ . La multiplication est effectuée en premier, bien qu'elle apparaisse en second dans la chaîne d'entrée.

Nous laissons au lecteur le soin de poursuivre les définitions des nombres et des noms<sup>1</sup>.

Dans les règles précédentes les caractères en gras (+, /, par exemple) sont des *symboles terminaux* du langage, ils ne nécessitent pas de définition syntaxique supplémentaire, seule la sémantique permet de savoir à quelles actions ils correspondent. Le formalisme BNF emploie quelques symboles bizarres dont voici l'explication :

- ::= signifie « est défini par » ;
- { } signifie « n'importe quelle séquence de zéro ou plusieurs occurrences des éléments encadrés par les accolades » ;
- [ ] signifie « zéro ou une occurrence des éléments encadrés par les crochets » ;
- | représente « ou » au sens exclusif du terme.

Sous des apparences simplistes, le formalisme BNF est extrêmement riche. On notera, en particulier, que nos règles contiennent la gestion des priorités d'opérateurs ; que le traitement des parenthèses se fait par une remontée récursive à la règle initiale, ce qui supprime toute limite au nombre des parenthèses imbriquées.

L'aspect non contextuel d'une grammaire passe par l'existence de mots réservés (les symboles syntaxiques terminaux), outre les symboles d'opérateurs (+, - ...) VHDL comporte près d'une centaine de mots réservés (if, then, case, etc.). Dans les descriptions syntaxiques nous utiliserons des caractères gras pour les mots réservés du langage. Dans les programmes eux mêmes, ces mots ne doivent bien évidemment pas être mis en caractères gras !

Dans le manuel de référence, les classes syntaxiques ou *productions*, sont classées par ordre alphabétique. C'est une méthode qui rend la recherche d'une construction très rapide ... à condition de savoir par où commencer. Nous conserverons donc dans les définitions syntaxiques la terminologie d'origine, en nous limitant à un sous-ensemble de la syntaxe complète du langage.

## 2.2 PREMIER APERÇU

La syntaxe est au service du sens. Nous rentrons ici dans le vif du sujet.

VHDL sert à décrire des circuits, ne l'oubliez jamais ! La source d'erreurs principale, pour les débutants, est de se laisser bercer par des constructions algorithmiques qui auraient un sens en C ou en PASCAL, mais qui ne correspondent à aucun circuit réalisable.

VHDL sert à modéliser des systèmes numériques à un niveau purement fonctionnel, il sert également à synthétiser, c'est-à-dire à traduire en un schéma logique, ces mêmes systèmes. Certaines constructions du langage sont de purs outils de modélisation, utilisables pour simuler le fonctionnement d'un système, elles ne sont pas toutes synthétisables. Ce premier aperçu va tenter de placer la scène et de donner un exemple de la différence qui peut exister entre modélisation et synthèse.

1. La poursuite de cette description fournit le squelette d'un programme de calculatrice. Le lecteur intéressé consultera avec profit A. Aho & al. , *Compilateurs principes, techniques et outils*, InterEditions, 1990 ; ou B. Stroustrup, *Le langage C++*, InterEditions, 1989.

### 2.2.1 L'exemple incontournable : une commande de feux tricolores

Rares sont les livres traitant d'électronique numérique qui ne proposent pas une version de feu rouge, respectons donc la tradition. Dans cet exemple nous sommes évidemment obligés d'utiliser des instructions que nous n'avons pas encore étudiées, nous tenterons de les garder suffisamment simples pour que leur sens soit évident.

#### a) Ce que l'on veut obtenir

Un chemin de campagne coupe une route à grande circulation. Les flots de véhicules qui franchissent le carrefour sont contrôlés par un feu tricolore :

- Par défaut le feu est au vert pour la grande route, et au rouge pour le chemin.
- Quand arrive un véhicule sur le chemin, un détecteur de passage provoque le changement des feux pour 30 secondes, puis les feux reviennent à la situation par défaut.
- Si plusieurs véhicules se suivent sur le chemin de campagne, il ne faut pas bloquer la grande route. Le retour à la situation par défaut au bout des 30 secondes est donc inconditionnel, indépendant de l'état du détecteur de passage. De plus, la situation par défaut doit durer au moins 1 minute.
- Le passage d'une situation à l'autre se fait par une séquence de sécurité qui prend 20 secondes : 10 secondes de feux orange d'un côté et rouge de l'autre, 10 secondes de feux rouges des deux côtés.

Dire ce que l'on veut faire fournit presque la solution. Traduisons ce qui précède par un synoptique et un diagramme de transitions (figure 2-1). Précisons que nous sommes encore très loin des circuits électroniques, nous ne nous préoccupons donc pas encore d'horloges ou de bascules.

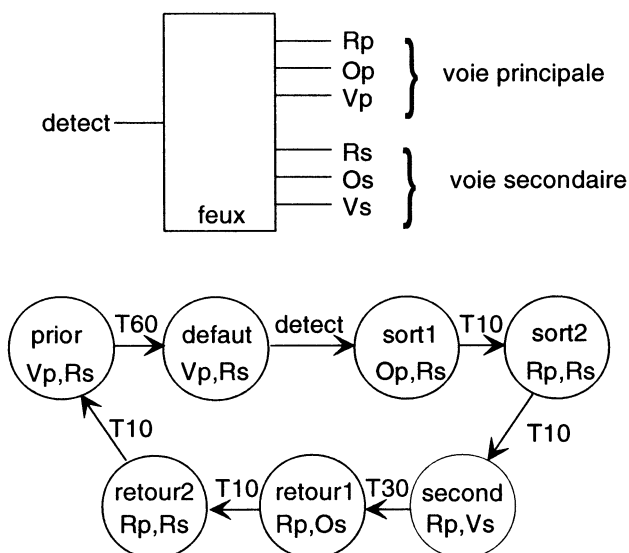


Figure 2-1 Une vue fonctionnelle de la commande de feux

Sur le diagramme de transitions de la figure 2-1 chaque état porte un nom (première ligne) et rend actives les sorties indiquées (deuxième ligne). Les sorties non mentionnées dans un état sont inactives, les feux correspondants sont éteints. Les noms des sorties rappellent leurs rôles : Rp pour feu rouge sur la voie principale, Rs pour son homologue de la voie secondaire, etc.

À côté de chaque transition figure une condition qui doit être vraie pour que cette transition puisse être effectuée.

L'ensemble synoptique-diagramme de transitions est suffisamment simple pour que l'on en déduise directement un programme VHDL dont voici une version :

```
-- feu_fonc.vhd
-- exemple fonctionnel non synthétisable

entity feux is
  port ( detect : in bit ;
         Rp, Op, Vp, Rs, Os, Vs : out bit );
end feux ;

architecture vue_esprit of feux is
  type etat_feu is (default, prior, sort1,
                   sort2, second, retour1, retour2);
  constant T10 : time := 10 sec ;
  constant T30 : time := 30 sec ;
  constant T60 : time := 60 sec ;
  signal automate : etat_feu ;
begin

  sorties : process
  begin
    Rp <= '0' ; Op <= '0' ; Vp <= '0' ;
    Rs <= '0' ; Os <= '0' ; Vs <= '0' ;
    case automate is
      when default => Vp <= '1' ; Rs <= '1' ;
      when sort1   => Op <= '1' ; Rs <= '1' ;
      when sort2   => Rp <= '1' ; Rs <= '1' ;
      when second  => Rp <= '1' ; Vs <= '1' ;
      when retour1 => Rp <= '1' ; Os <= '1' ;
      when retour2 => Rp <= '1' ; Rs <= '1' ;
      when prior   => Vp <= '1' ; Rs <= '1' ;
    end case ;
    wait on automate ;
  end process sorties ;

  sequence : process
  begin
    case automate is
      when default => if detect = '0' then
        wait until detect = '1' ;
      end if ;
      automate <= sort1 ;
      wait on automate ;
    end case ;
  end process sequence ;
end architecture vue_esprit of feux ;
```

```

when sort1 => wait for T10 ;
    automate <= sort2 ;
    wait on automate ;
when sort2 => wait for T10 ;
    automate <= second ;
    wait on automate ;
when second => wait for T30 ;
    automate <= retour1;
    wait on automate ;
when retour1 => wait for T10 ;
    automate <= retour2;
    wait on automate ;
when retour2 => wait for T10 ;
    automate <= prior ;
    wait on automate ;
when prior => wait for T60 ;
    automate <= default ;
    wait on automate ;
end case ;
end process sequence ;
end vue_esprit ;

```

Dans cet exemple nous n'avons pas cherché à minimiser la longueur du programme, préférant refléter strictement, dans le code source, la structure du diagramme de transitions. Le fonctionnement se comprend bien si on réalise que :

- chaque processus est une boucle sans fin dont le code s'exécute séquentiellement ;
- les deux processus s'exécutent simultanément, en parallèle : le processus « sequence » suit le diagramme de transitions tandis que le processus « sorties » calcule les valeurs des commandes des feux en fonction de l'état du précédent ;
- la modification de la valeur d'un signal ne prend effet que quand le simulateur rencontre la fin du processus ou une instruction `wait`, par défaut un signal conserve sa valeur initiale ;
- un processus peut se mettre en attente d'un événement (modification de la valeur d'un signal) ou d'un temps « simulateur », grâce à l'instruction `wait`.

L'examen rapide d'un résultat de simulation (simulateur V-system de la société Model Technology), figure 2-2, nous fournit une présomption de preuve de la correction de notre programme.

Un cycle complet de blocage de la voie principale dure bien 70 secondes ; si une file de véhicules occupe la voie secondaire, notre algorithme assure bien une alternance des feux verts au rythme de 60 secondes pour la voie principale et 30 secondes pour la voie secondaire.

Pour passer à un modèle synthétisable il nous faut quelque peu étoffer certains points de notre solution, c'est l'objet du paragraphe suivant.

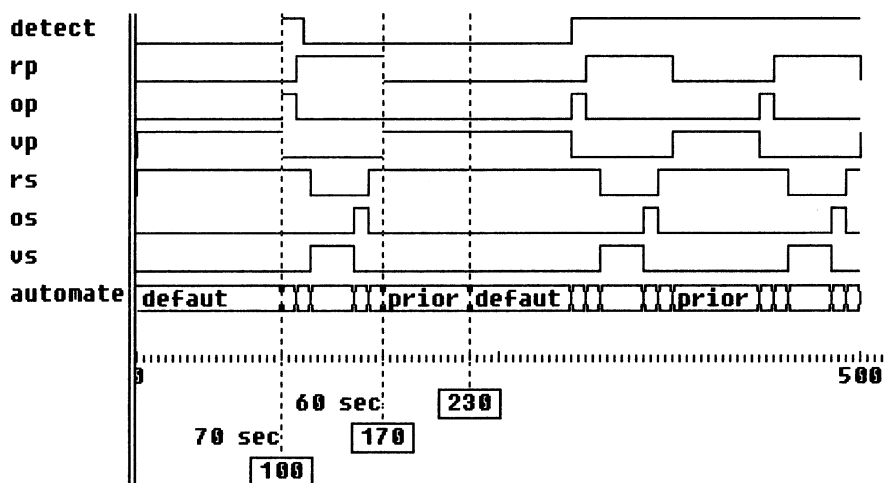


Figure 2-2 Simulation fonctionnelle de la commande de feux.

### b) La façon de le réaliser

Le modèle de commande de feux précédent passe pudiquement sous silence deux aspects :

- Comment réalise-t-on les temporisations T10, T30 et T60 ? Trois signaux sont créés à cet effet, pilotés par une temporisation qui est réinitialisée quand l'automate est dans l'état défaut.
- Comme tout système séquentiel qui se respecte, le fonctionnement de l'automate doit être une machine d'états synchrone ; il faut donc prévoir un signal d'horloge. Pour ne pas compliquer inutilement la solution, nous admettrons qu'une horloge de fréquence 1Hz est disponible.

Le synoptique de la figure 2-3 reflète ces modifications ; le diagramme de transitions n'a pas de raison d'être modifié.

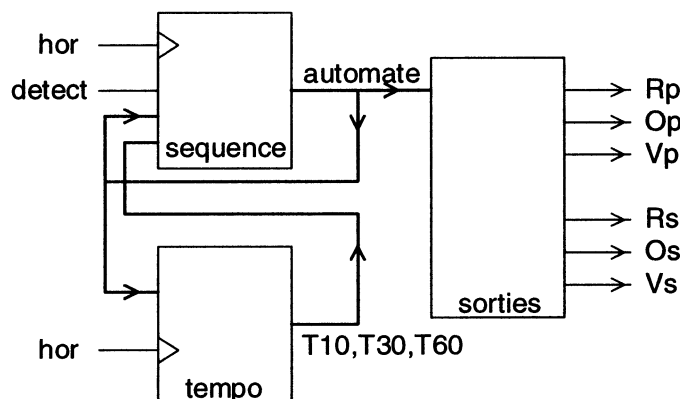


Figure 2-3 Commande de feux plus réaliste.

Un bloc de temporisation naïf peut être obtenu par un simple compteur, capable de compter 130 périodes d'horloge<sup>1</sup> (somme de tous les temps d'attente), qui est remis à zéro quand l'automate est dans l'état default.

D'où un programme VHDL synthétisable :

```
-- feu_synt.vhd

-- exemple synthétisable

entity feux is
  port ( hor, detect : in bit ;
        Rp, Op, Vp, Rs, Os, Vs : out bit );
end feux ;

architecture synthese of feux is
  type etat_feu is (default, prior, sort1,
                   sort2, second, retour1, retour2);
  signal T10, T30, T60 : bit ;
  signal temps : integer range 0 to 129 ;
  signal automate : etat_feu ;
begin

  sorties : process
  begin
    Rp <= '0' ; Op <= '0' ; Vp <= '0' ;
    Rs <= '0' ; Os <= '0' ; Vs <= '0' ;
    case automate is
      when default => Vp <= '1' ; Rs <= '1' ;
      when sort1   => Op <= '1' ; Rs <= '1' ;
      when sort2   => Rp <= '1' ; Rs <= '1' ;
      when second  => Rp <= '1' ; Vs <= '1' ;
      when retour1 => Rp <= '1' ; Os <= '1' ;
      when retour2 => Rp <= '1' ; Rs <= '1' ;
      when prior   => Vp <= '1' ; Rs <= '1' ;
    end case ;
    wait on automate ;
  end process sorties ;

  sequence : process
  begin
    wait until hor = '1' ;
    case automate is
      when default => if detect = '1' then
        automate <= sort1 ;
      end if ;
      when sort1   => if T10 = '1' then
        automate <= sort2 ;
      end if ;
```

1. Solution évidemment stupide si on se pose la question du nombre de bascules utilisées, mais ce n'est pas notre propos ici.

```

    when sort2 => if T10 = '1' then
        automate <= second ;
    end if ;
    when second => if T30 = '1' then
        automate <= retour1;
    end if ;
    when retour1 => if T10 = '1' then
        automate <= retour2;
    end if ;
    when retour2 => if T10 = '1' then
        automate <= prior ;
    end if ;
    when prior => if T60 = '1' then
        automate <= default ;
    end if ;
end case ;
end process sequence ;

tempo : process
begin
    wait until hor = '1' ;
    if automate = default or temps = 129 then
        temps <= 0 ;
    else
        temps <= temps + 1 ;
    end if ;
end process tempo ;

calcul_Ti : process
begin
    T10 <= '0' ; T30 <= '0' ; T60 <= '0' ;
    case temps is
        when 9 => T10 <= '1' ;
        when 19 => T10 <= '1' ;
        when 49 => T30 <= '1' ;
        when 59 => T10 <= '1' ;
        when 69 => T10 <= '1' ;
        when 129 => T60 <= '1' ;
        when others => null ;
    end case ;
    wait on temps ;
end process calcul_Ti ;

end synthese ;

```

Vu de loin le résultat de simulation est identique au précédent, ce qui est rassurant. En examinant le chronogramme de près, cependant, on constaterait que, quand le signal detect est toujours actif, l'état default de l'automate dure une période d'horloge, au lieu d'un temps nul dans le modèle purement fonctionnel. Cette durée minimum est caractéristique d'un système synchrone.



Concernant la description de l'automate, la différence majeure réside dans le mécanisme d'attente du processus correspondant : seul un événement relatif à l'horloge est susceptible de le faire évoluer. Pourquoi les processus *sorties* et *calcul\_Ti* génèrent ils des fonctions combinatoires, alors que les processus *sequence* et *tempo* génèrent des machines d'états synchrones, est une histoire qui mérite une attention toute particulière, mais n'anticipons pas. Nous reviendrons en détail sur ces remarques importantes.

### 2.2.2 Le couple *entity architecture*

Un opérateur élémentaire, un circuit intégré, une carte électronique ou un système complet est complètement défini par des signaux d'entrées et de sorties et par la fonction réalisée de façon interne. Ce double aspect - signaux de communication et fonction - se retrouve à tous les niveaux de la hiérarchie d'une application. L'élément essentiel de toute description en VHDL, nommé *design entity* dans le langage, est formé par le couple *entity - architecture*, qui décrit l'apparence externe d'une unité de conception et son fonctionnement interne.

#### a) La boîte noire : une entité

La déclaration d'entité décrit l'interface entre le monde extérieur et une unité de conception : signaux d'entrées et de sorties et, éventuellement, paramètres génériques (i.e. constantes dont la valeur est fixée par l'environnement de l'unité considérée). La déclaration d'entité peut également contenir des instructions concurrentes *passives*, c'est-à-dire qui ne contiennent aucune affectation de signaux ; de telles instructions ne génèrent pas de circuit lors du processus de synthèse.

Une même entité peut être associée à plusieurs architectures différentes. Elle décrit alors une classe d'unités de conception qui présentent au monde extérieur le même aspect, avec des fonctionnements internes différents. Le même circuit peut, par exemple, être décrit de façon purement fonctionnelle lors de la conception, et de façon structurelle pour contrôler le bon respect des règles temporelles de ses constituants physiques.

#### » Syntaxe

```
entity_declaration ::=
entity identifiant is
[ generic ( generic_list ) ; ]
[ port ( port_list ) ; ]
entity_declarative_part
[ begin
entity_statement_part ]
end [ entity ] [ identifiant ] ;
```

La zone déclarative (*entity\_declarative\_part* ) et la zone d'instructions (*entity\_statement\_part* ) contiennent des informations qui sont connues de toutes les architectures qui partagent la même entité. Nous n'en détaillerons pas ici les contenus, certains exemples illustreront leur utilisation éventuelle. Notons

cependant qu'en ce qui concerne les instructions, seules des instructions « passives », c'est-à-dire qui ne provoquent aucun changement de signal, sont légales dans l'entité. Il est, par exemple, possible de vérifier à cet endroit que des spécifications électriques concernant les signaux d'entrées, comme une largeur minimum d'impulsion, sont respectées.

Rappelons que, dans ces descriptions syntaxiques, les mots réservés du langage sont indiqués en caractères gras et que les éléments entre crochets ne sont pas obligatoires.

### ► Exemples

Une déclaration d'entité pour un multiplexeur de quatre voies vers une voie :

```
entity mux4_1 is
  port( e0,e1,e2,e3 : in bit ;
        sel : in bit_vector(1 downto 0) ;
        sort : out bit ) ;
end mux4_1 ;
```

Une déclaration d'entité pour un multiplexeur de dimension arbitraire<sup>1</sup> :

```
entity muxN_1 is
  generic ( dimension : integer := 4 ) ;
  port( entree : in bit_vector(dimension-1 downto 0) ;
        sel : in integer range 0 to dimension-1 ;
        sort : out bit ) ;
end muxN_1 ;
```

Une déclaration d'entité pour un registre bidirectionnel de dimension arbitraire:

```
entity registre_N is
  generic ( dimension : integer := 8 ) ;
  port(hor, direction : in bit ;
        donnee : inout std_logic_vector(0 to dimension-1));
end registre_N ;
```

### ► Ports d'accès

La liste des ports d'accès (**port** ( port\_list ) ; ) déclare l'ensemble des signaux d'interface entre le monde extérieur et l'unité décrite. À chaque signal de communication sont attachés un nom, un type et un mode :

```
port_list ::=
  identifieur_list : [ mode ] subtype_indication
  { ; identifieur_list : [ mode ] subtype_indication }
  identifieur_list ::= identifieur { , identifieur }
  mode ::= in | out | inout | buffer | linkage
```

1. Nous verrons que ce paramètre de dimension, qui vaut ici 4 par défaut, peut être modifié, par exemple quand on souhaite utiliser l'opérateur comme composant instancié dans un ensemble plus complexe. Les différentes instances du multiplexeur peuvent alors avoir des dimensions différentes.

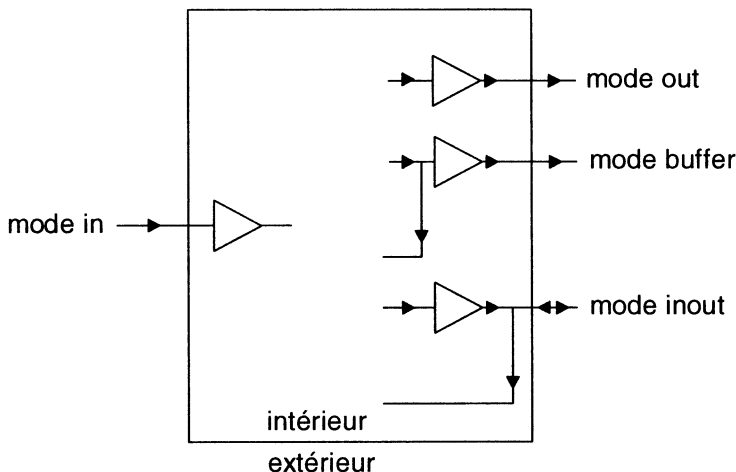
Le nom (identifier) est connu à l'intérieur de toutes les architectures qui font référence à l'entité correspondante. Il est constitué par une chaîne de caractères alphanumériques qui commence par une lettre et qui peut contenir le caractère souligné ( \_ ); VHDL ne distingue pas les majuscules des minuscules, AmI et aMi représentent donc le même objet.

Le mode précise le sens de transfert des informations<sup>1</sup>, s'il n'est pas précisé le mode in est supposé. Les modes in et out sont simples à interpréter : les informations transitent par les ports correspondants comme sur des voies à sens unique. Ils correspondent, dans le cas d'un circuit, aux broches d'entrées et de sorties des informations ; un circuit ne peut évidemment pas modifier de lui même les valeurs de ses signaux d'entrées. Il ne peut pas non plus, et cela est moins évident, « relire » la valeur de ses signaux de sortie.

Le mode buffer décrit un port de sortie dont la valeur peut être « relue » à l'intérieur de l'unité décrite. Il correspond à un signal de sortie associé à un rétrocouplage vers l'intérieur d'un circuit, par exemple.

Le mode inout décrit un signal d'interface réellement bidirectionnel : les informations peuvent transiter dans les deux sens. Il faut bien sûr, dans ce cas, prévoir la gestion de conflits potentiels, la valeur du signal présent sur le port pouvant avoir deux sources différentes.

Les deux modes précédents ne doivent pas être confondus, le premier correspond à un signal dont la source est unique, interne à l'unité décrite, le second correspond à un signal dont les sources peuvent être multiples, internes et externes : un bus. Dans la figure 2-4 des flèches illustrent, pour chaque mode, les sens de transfert des informations entre l'intérieur d'une unité et l'extérieur.



**Figure 2-4** Ports d'interfaces et modes.

1. Sauf le mode linkage, qui sert de lien entre différents niveaux d'une construction hiérarchique sans rien préciser de plus. Il est fort peu utilisé en pratique.

Le type des données transférées doit être précisé. Comme ces données sont échangées avec le monde extérieur, les définitions de leurs types doivent être faites à l'extérieur de l'entité ; il est donc logique que la zone de déclaration locale se trouve après celle qui définit les ports, cela n'aurait aucun sens de vouloir l'utiliser pour définir le type associé à un port, ce type serait inconnu de l'extérieur.

### ► Paramètres génériques

Les paramètres génériques sont des constantes qui permettent de créer des modules configurables au moment de leur utilisation. L'exemple le plus classique consiste à mettre sous forme générique des dimensions de données, nombre d'éléments binaires d'un bus ou profondeur d'une mémoire *fifo*, par exemple. Les paramètres génériques ne sont ni des variables ni des signaux, leurs valeurs doivent être connues au moment de la compilation finale d'un projet.

On notera que la déclaration de paramètres génériques précède celle des ports, ce qui permet de paramétrer ces derniers.

### b) Son contenu : une architecture

Le fonctionnement interne d'un module, son *corps*, est précisé par une architecture associée à l'entité qui décrit l'aspect extérieur de ce module. Une architecture porte un nom, ce qui autorise la création de plusieurs architectures différentes pour la même déclaration d'entité. Une unité de conception est la réunion d'une entité et d'une architecture.

### ► Syntaxe

L'architecture est divisée en deux parties : une zone déclarative et une zone d'instructions. Ces instructions sont concurrentes, elles s'exécutent en parallèle. Cela signifie que les instructions d'une architecture peuvent être écrites dans un ordre quelconque, le fonctionnement ne dépend pas de cet ordre d'écriture. Ce point est simple à comprendre si on « pense circuits », les différentes parties d'un circuit coexistent et agissent simultanément ; il peut être un peu surprenant pour les programmeurs habitués des langages procéduraux comme C ou PASCAL, qui continuent à « penser » algorithmes séquentiels.

```
architecture_body ::=
architecture architecture_name of entity_name is
  architecture_declarative_part
begin
  architecture_statement_part
end [ architecture ] [ architecture_name ] ;
```

La zone déclarative permet de définir des types, des objets (signaux, constantes) locaux au bloc considéré. Elle permet également de déclarer des noms d'objets externes (composants d'une librairie, par exemple) utilisés dans le corps de l'architecture.

### ➤ Exemples

Les trois exemples qui suivent correspondent aux trois exemples de déclarations d'entités donnés précédemment.

Un multiplexeur de quatre voies vers une voie :

```
architecture essai of mux4_1 is
    constant zero  : bit_vector(1 downto 0) := "00" ;
    constant un    : bit_vector(1 downto 0) := "01" ;
    constant deux  : bit_vector(1 downto 0) := "10" ;
    constant trois : bit_vector(1 downto 0) := "11" ;
begin
    process (e0,e1,e2,e3,sel)
    begin
        case sel is
            when zero  => sort <= e0 ;
            when un    => sort <= e1 ;
            when deux  => sort <= e2 ;
            when trois => sort <= e3 ;
        end case ;
    end process ;
end essai ;
```

Notons en passant que <= représente l'opérateur d'affectation d'un signal.

Un multiplexeur de dimension arbitraire :

```
architecture essai of muxN_1 is
begin
    sort <= entree(sel) ;
end essai ;
```

Un registre bidirectionnel de dimension arbitraire :

```
architecture essai of registre_N is
    signal temp : std_logic_vector(dimension - 1 downto 0) ;
begin
    donnee <= temp when direction = '0' else (others => 'Z') ;1
    process
    begin
        wait until hor = '1' ;
        if direction = '1' then
            temp <= donnee ;
        end if ;
    end process ;
end essai ;
```

On distingue classiquement en VHDL trois styles de descriptions, qui peuvent être utilisés simultanément.

1. L'expression (others => 'Z') est un agrégat. La même valeur 'Z' est affectée à tous les éléments du tableau.

### ➤ Description comportementale

Une description comportementale (*behavioral*) se présente comme des blocs d'algorithmes séquentiels exécutés par des processus indépendants. Elle peut traduire un fonctionnement séquentiel ou combinatoire du circuit modélisé. Nous expliciterons ce point en détail dans la suite.

Un simple multiplexeur deux voies vers une voie peut être décrit par un algorithme qui reproduit son comportement :

```
entity mux2_1 is
    port(e0,e1 : in bit ;
          sel : in bit ;
          sort : out bit ) ;
end mux2_1 ;

architecture comporte of mux2_1 is
begin
    process (e0,e1,sel)1
    begin
        if sel = '0' then
            sort <= e0 ;
        else
            sort <= e1 ;
        end if ;
    end process ;
end comporte ;
```

### ➤ Description flot de données

Une description flot de données (*data flow*) correspond grosso modo à un *register transfert language*, les signaux passent à travers des couches d'opérateurs logiques qui décrivent les étapes successives qui font passer des entrées d'un module à sa sortie. Le même multiplexeur élémentaire s'écrit, dans ce style :

```
architecture flot of mux2_1 is
    signal sele0, sele1 : bit ;
begin
    sort <= sele0 or sele1 ;
    sele0 <= e0 and not sel ;
    sele1 <= e1 and sel ;
end flot ;
```

### ➤ Description structurelle

Une description structurelle (*structural*) utilise des composants supposés exister dans une librairie de travail, sous forme d'unités de conception. Le programme se contente alors d'*instancier* les composants nécessaires et de décrire leurs interconnexions.

1. Les éléments mis entre parenthèse indiquent les signaux dont les changements doivent « réveiller » le processus, et donc provoquer l'évaluation du signal de sortie.

Le même multiplexeur utilise deux portes ET, un NON et un OU<sup>1</sup> :

```
architecture struct of mux2_1 is
  component et
    port (a, b : in bit ; s : out bit) ;
  end component ;
  component ou
    port (a, b : in bit ; s : out bit) ;
  end component ;
  component non
    port (a : in bit ; s : out bit) ;
  end component ;
  signalonsel, sele0, sele1 : bit ;
begin
  result : ou port map(sele0, sele1, sort) ;
  complem : non port map(sel,onsel) ;
  choix0 : et port map(onsel, e0, sele0) ;
  choix1 : et port map(sel, e1, sele1) ;
end struct ;
```

Ce dernier programme suppose que les composants portent les mêmes noms que les unités de conception auxquelles ils se réfèrent, ce n'est en rien une obligation. Des déclarations de configuration permettent de créer des liens, entre les composants instanciés et les couples entité architecture, autres que par homonymie.

Les exemples qui précèdent sont d'une naïveté qui n'utilise pas la puissance du langage, il ne resterait rien de ces programmes si on cherchait à rendre le code source plus compact, c'est une évidence.

Le plus souvent, on utilisera une description structurelle au niveau supérieur d'un projet, et des descriptions des deux autres types, suivant les fonctions décrites et les goûts du programmeur, pour les modules instanciés. Les programmes VHDL générés par les outils de placement routage, à des fins de vérification temporelle du bon fonctionnement d'une application, sont bien sûr essentiellement structurels : ils reproduisent le câblage réellement effectué dans le circuit ou sur une carte. Les fondeurs fournissent des modèles comportementaux des opérateurs élémentaires de leurs circuits qui prennent en compte leurs caractéristiques dynamiques, temps de propagation, entre autres.

### 2.2.3 Types et classes

VHDL est un langage fortement typé, tout objet<sup>2</sup> manipulé doit avoir un type défini avant la création de l'objet. Indépendamment de son type, un objet appartient à une classe<sup>3</sup>. Schématiquement, on peut dire que le type définit le format des données et

1. Le lecteur est vivement convié à dessiner le schéma classique d'un tel multiplexeur.

2. Dans le langage un objet est défini comme une grandeur nommée qui contient une valeur d'un type défini. Une *entity*, par exemple, n'est donc pas un objet. Le fait qu'un objet doive être nommé souffre quelques exceptions.

3. En réalité cette indépendance entre classe et type n'est pas vérifiée pour les fichiers : la classe *file* est associée à des types qui lui sont spécifiques. La manipulation des fichiers sera abordée plus loin dans ce chapitre, à propos des outils de modélisation.

l'ensemble des opérations légales sur ces données, alors que la classe définit un comportement dynamique, précise la façon dont évolue (ou n'évolue pas dans le cas d'une constante !) une donnée au cours du temps.

Le langage distingue quatre catégories de types :

- Les types scalaires, c'est-à-dire les types numériques et énumérés, qui n'ont pas de structure interne.
- Les types composés (tableaux et enregistrement) qui possèdent des sous-éléments.
- Les types *access*, qui sont des pointeurs.
- Le type *file*, qui permet de gérer des fichiers séquentiels.

À partir d'un type il est possible de définir un sous-type par l'adjonction d'une contrainte et/ou d'une *fonction de résolution*<sup>1</sup>. Une contrainte est la restriction des valeurs possibles à un domaine plus limité que celui qui définit le type de base (entiers de 0 à 255, par exemple). Un sous-type hérite des opérations définies sur le type de base, mais le contrôle du respect de la contrainte sur le résultat est à la charge du programmeur (200 + 200 n'est pas compris entre 0 et 255).

Outre les types prédéfinis, l'utilisateur peut créer ses propres types :

### ➤ Syntaxe générale

```
full_type_declaration ::=
type identifier is type_definition ;
type_definition ::=
scalar_type_definition | composite_type_definition
| access_type_definition | file_type_definition

subtype_declaration ::=
subtype identifier is [resolution_function_name ]
type_mark [constraint ] ;
```

Notons que du point de vue syntaxique, un type est sous-type de lui-même, cette remarque intervient dans la compréhension de certaines règles syntaxiques décrites dans le manuel de référence.

### ➤ Exemples

```
type octet is range 0 to 255 ;
type mem8 is array (0 to 1023) of octet ;
FUNCTION resolved ( s : std_ulogic_vector )
RETURN std_ulogic;
SUBTYPE std_logic IS resolved std_ulogic;
subtype word is integer range -32768 to 32767 ;
```

1. Les fonctions de résolutions permettent de modéliser les structures de bus où plusieurs sorties sont connectées en parallèle. Elles seront étudiées au §7 de cette partie.



### a) Les types scalaires

Les types scalaires sont définis par une liste de valeurs (types énumérés) ou par un domaine de définition (types numériques). VHDL est un langage de description de circuits (numériques pour l'instant) ; le compilateur doit pouvoir déduire de la définition d'un type sa taille en nombre de chiffres binaires, et limiter au mieux cette taille. Un additionneur 8 bits n'est pas du même ordre de grandeur de complexité qu'un additionneur 64 bits ! Cela explique l'importance accordée aux domaines de définition par ce langage.

Un domaine de définition (range) est déterminé généralement par la spécification de bornes et un sens (to ou downto) de parcours, ascendant ou descendant. Il est également possible d'utiliser un attribut qui fait référence à un type tableau (ou à un tableau) de dimension connue, dont on veut « copier » la dimension. Parmi les types numériques, la différence entre entiers et flottants est faite à partir de l'expression des bornes :

```
type octet is range 0 to 255 ;
type valim is range 0.0 to 5.0 ;-- non synthétisable
type mem8 is array (0 to 1023) of octet ;
type adresse is range mem8'range ;
```

Les types scalaires sont tous ordonnés, les opérateurs relationnels leur sont donc applicables sans autre précaution. Les types entiers et énumérés sont dits discrets, nous verrons qu'ils peuvent intervenir comme indices de tableaux.

### » Types énumérés

Un type énuméré est construit par une liste de valeurs de type caractère ou par une liste d'identificateurs :

```
type ami is (mzero,pzero,moins,plus) ;
type bit is ('0','1') ;
type boolean is (FALSE,TRUE) ;

TYPE std_ulogic IS ( 'U', -- Uninitialized
                    'X', -- Forcing Unknown
                    '0', -- Forcing 0
                    '1', -- Forcing 1
                    'Z', -- High Impedance
                    'W', -- Weak Unknown
                    'L', -- Weak 0
                    'H', -- Weak 1
                    '-' -- Don't care
                );
```

À chaque élément de la liste, qui définit un type énuméré, est attaché un nombre entier qui identifie sa position. La numérotation est faite de gauche à droite en partant de zéro. Ces positions induisent une relation d'ordre. Par exemple : TRUE est supérieur à FALSE.

Les types prédéfinis du langage sont les types bit et boolean, sur lesquels sont définies les opérations logiques élémentaires, le type severity\_level lié à l'instruction assert, et deux types liés aux manipulations de fichiers : file\_open\_kind et file\_open\_status.

### ➤ Types entiers

Les types entiers sont simplement définis par leur domaine de variation (range). Les opérations arithmétiques habituelles sont définies sur tous les types entiers. Au cours d'une simulation, si une opération provoque un débordement, une erreur est générée ; c'est au programmeur de veiller à ce que de telles situations ne se produisent pas.

Le type entier prédéfini est le type `integer`. Son domaine de variation correspond au moins à un nombre codé en binaire sur 32 bits (-2147483648 à 2147483647).

### ➤ Types physiques

Les types physiques décrivent des objets qui peuvent prendre des valeurs entières, exprimées avec des unités définies par le type :

```
physical_type_declaration ::=
    type physical_type_name is
    range_constraint
    units
        prim_unit_name ;
        { sec_unit_name = [ constante ] unit_name ;
        end units [ physical_type_name ] ;
```

Le type physique prédéfini est le type `time` :

```
type time is range -2_147_483_647 to 2_147_483_647
    units
        fs;
        ps = 1000 fs;
        ns = 1000 ps;
        us = 1000 ns;
        ms = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hr = 60 min;
    end units;
```

Son domaine de définition dépend des implémentations, mais ne peut pas être inférieur à celui indiqué dans la déclaration ci-dessus. Ce type est évidemment très utile dans des programmes de simulation.

### ➤ Types flottants

Inutilisables actuellement en synthèse des systèmes numériques, les types flottants servent à la modélisation. Leur utilisation pour représenter des fonctionnements analogiques est encore embryonnaire. Les systèmes de CAO ne fournissent pas encore tous les bibliothèques mathématiques associées. Notons que ces bibliothèques, quand elles existent, sont très complètes et contiennent toutes les fonctions que l'on rencontre habituellement dans les langages de programmation.

Le type flottant prédéfini du langage est le type `real`, décrit par :

```
type real is range -1.0E38 to 1.0E38 ;
```

Normalement le domaine de définition ne devrait pas être inférieur à celui indiqué ci-dessus, et la précision ne devrait pas être inférieure à 6 chiffres décimaux. Certains compilateurs, bien que connaissant le type, ne respectent pas ces contraintes.

### b) Les types structurés

Comme tous les langages de haut niveau, VHDL permet à l'utilisateur de structurer ses données au moyen des types composites que sont les tableaux (*arrays*) et les enregistrements (*records*). Ces types sont constitués d'éléments, scalaires ou eux-mêmes composites, tous de même type dans le cas des tableaux, de types différents dans le cas des enregistrements.

#### » Tableaux

Un tableau est une collection d'éléments, tous de même type, qui sont repérés par les valeurs d'indices. Un tableau caractérisé par un seul indice (cas fréquent) est dit unidimensionnel, on parle alors d'un vecteur. Les indices doivent être des objets de types scalaires discrets, donc entiers ou énumérés. Notons, en passant, que cette dernière possibilité permet de construire très facilement des tables de vérité.

Lors de la définition d'un type, il est possible de ne pas spécifier tout de suite les bornes des indices, constituant par là un tableau générique *non contraint* (*unconstrained*). Des sous-types peuvent alors dériver du type non contraint en précisant simplement les bornes et le sens de parcours des valeurs des indices. Cette spécification peut se faire par la création d'un sous-type nommé explicitement ou simplement lors de la déclaration d'un objet. C'est de cette façon qu'est construit le type `bit_vector` dans la librairie standard. Il est clair que lors de la compilation tous les objets de type tableau doivent avoir une taille connue, donc être de type tableau contraint.

#### Syntaxe générale :

```
array_type_declaration ::=
    type nom_tableau is
        array ( index_def { , index_def } ) of element_subtype ;
index_def ::=
    discrete_range | type_mark range <>
```

#### Exemples :

```
subtype natural is integer range 0 to integer'high;
type bit_vector is array (natural range <>) of bit;

type table_verite_2 is array(bit,bit) of bit ;
constant ouex_tv : table_verite_2 := (('0','1'),('1','0')) ;
```

```
-- un tableau à deux dimensions est un tableau de vecteurs
-- d'où la construction peu évidente de la valeur
constant ouex_tv1 : table_verite_2 := ("01","10") ;
```

Les habitués du langage C seront agréablement surpris : VHDL manipule les objets de type tableau de façon globale, par leur nom, tout simplement<sup>1</sup>.

Outre le type `bit_vector`, d'un usage des plus courants, le langage connaît le type `string`, pour chaîne de caractères, et, moyennant l'utilisation de la librairie IEEE dont nous reparlerons, le type `std_logic_vector`<sup>2</sup> :

```
subtype positive is integer range 1 to integer'high;
type string is array (positive range <>) of character;
TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF
    std_logic;
```

### » Enregistrements

Équivalents des structures de C, les enregistrements (*records*) permettent de rassembler des objets de types différents dans une même organisation, et de repérer chaque élément par son nom.

*Syntaxe :*

```
record_type_declaration ::=
    type record_name is
        record
            element_declaration
            { element_declaration }
        end record [ record_name ] ;
```

*Exemple :*

```
type nom_de_mois is (jan,fev,mar,avr,mai,jun,
                    jul,aou,sep,oct,nov,dec) ;

type date is
    record
        jour : integer range 0 to 31 ;
        mois : nom_de_mois ;
        annee : integer range -2000 to 4000 ;
    end record date ;
```

### c) Pointeurs et fichiers

Les types pointeur (*access*) et fichiers (*file*) ne correspondent évidemment à aucun objet synthétisable dans un circuit. Pour le second c'est une évidence, pour le premier il suffit de préciser que leur manipulation suppose l'existence d'un allocateur de mémoire dynamique pour que les choses soient claires.

1. Ce que ne fait pas le C, qui sur ce point n'est pas particulièrement souple. Une fois n'est pas coutume.  
2. Ce type est construit à partir du type énuméré `std_ulogic`, qui a été défini précédemment.

Nous nous contenterons ici de donner la syntaxe de définition de ces types. Nous reviendrons plus loin sur la manipulation de fichiers dans le cadre des programmes de tests<sup>1</sup>.

### ➤ Pointeurs

```
access_type_declaration ::=
type nom_type is access type_pointe ;
```

Faute d'appel à un allocateur, un objet du type pointeur prend la valeur par défaut égale à null.

### ➤ Fichiers

Un fichier représente une séquence d'objets de même type, contenus dans un fichier du système hôte. Lors de la déclaration d'un type fichier des procédures d'accès sont déclarées implicitement : FILE\_OPEN(...), FILE\_CLOSE(...), READ(...), WRITE(...) et une fonction qui indique la fin d'un fichier, ENDFILE(...).

```
type nom_de_type is file of type_des_elements ;
```

L'accès au contenu d'un objet de type file se fait de façon séquentielle.

### d) Constantes signaux et variables

Un objet dans un programme VHDL appartient à l'une des quatre classes connues du langage, classe qui est précisée lors de la déclaration (obligatoire) de l'objet :

```
objet_declaration ::=
    constant_declaration
    | signal_declaration
    | variable_declaration
    | file_declaration
```

Nous abordons ici les trois premières classes, l'utilisation des fichiers sera vue ultérieurement.

Plusieurs objets de la même classe et du même type peuvent être déclarés simultanément au moyen d'une liste d'identificateurs.

### ➤ Constantes

Les constantes permettent de paramétrer sous forme littérale des valeurs non modifiables au cours de l'exécution d'un programme. Elles sont généralement initialisées au moment de leur déclaration<sup>2</sup>. Cette déclaration se situe dans la partie déclarative

1. Telle qu'est définie leur sémantique, qui passe par des allocateurs de mémoire, on peut se demander si l'introduction des pointeurs dans le langage était une nécessité absolue dans le cadre de la description de circuits. L'exemple du manuel de référence qui est la classique liste doublement chaînée n'est guère convaincant.
2. Pour préciser : dans un paquetage il est possible de différer l'initialisation d'une constante. La constante est déclarée dans la déclaration du paquetage et sa valeur fournie dans le corps de celui-ci.

d'une entité, d'une architecture, d'un processus, d'un bloc, d'un paquetage ou d'un sous-programme.

*Syntaxe :*

```
constant_declaration ::=
    constant ident_list : subtype_indication [:= expression] ;
```

L'expression qui initialise la constante doit évidemment être elle-même constante et du même type.

*Exemples :*

```
constant age_capitaine : integer := 25 ;
type table_verite_2 is array(bit,bit) of bit ;
constant ouex_tv : table_verite_2 := (('0','1'),('1','0'));
constant ouex_tv1 : table_verite_2 := ("01","10") ;
```

### ► Signaux

Les signaux jouent un rôle central dans le langage. Ils correspondent, et eux seuls, aux canaux de communication présents dans un circuit. Un langage de modélisation de circuits doit respecter, entre autres, deux contraintes importantes :

Il doit être causal<sup>1</sup>, les changements d'états des équipotentielles d'un circuit, appelés *événements*, obéissent à une chaîne causes → effets qui doit être préservée quels que soient les paramètres dynamiques (retards ou autres) des opérateurs élémentaires du circuit.

Il doit être concurrent, tous les opérateurs agissent simultanément, l'ordre dans lequel le programmeur les introduit dans sa description ne doit pas avoir de conséquence sur le résultat d'une simulation ou d'une synthèse.

Si l'on n'y prend pas garde, par exemple quand on se lance dans l'écriture d'un simulateur logique, ces deux contraintes peuvent être contradictoires. Une conséquence importante de ces remarques est que le comportement d'un signal est fort différent de celui d'une variable ordinaire d'un langage de programmation comme le C. La confusion entre les comportements des variables et des signaux est sans doute l'une des sources d'erreurs la plus importante des programmeurs non avertis : compteurs qui semblent compter en simulation mais ne font rien lors de la synthèse, circuits combinatoires qui se transforment en circuits séquentiels lors de la synthèse, et réciproquement, etc.

La méthode utilisée en VHDL pour traiter le problème est d'introduire le concept de pilote (*driver*) pour gérer les modifications de valeurs d'un signal. Toute opération modifie l'état d'un driver, la valeur réelle prise par un signal dépend des états des drivers qui lui sont connectés. Nous reprendrons en détail cette discussion au

1. Il existe des simulateurs logiques qui ne sont pas causaux. Même achetés auprès de faiseurs ayant pignon sur rue. Bien sûr, pour les mettre en défaut il faut un peu leur proposer des situations ambiguës, c'est la moindre des choses. Un conseil aux programmeurs en herbe : il est plus facile de supposer au départ que tous les opérateurs possèdent un temps de retard non nul, mais cela manque d'élégance.

paragraphe suivant, mais une mise en garde s'imposait avant de poursuivre notre froide description syntaxique qui ne laisse pas transparaître l'aspect sémantique des choses.

Les ports d'une entité appartiennent à la classe des signaux, nous ne reviendrons pas ici sur leur déclaration pour nous concentrer sur les signaux déclarés explicitement.

Un signal se déclare dans l'univers concurrent : zones déclaratives d'une architecture (le cas le plus fréquent), d'une entité, d'un bloc ou d'un paquetage.

*Syntaxe :*

```
signal_declaration ::=  
    signal identifier_list : subtype_indication  
        [register | bus ] [ := expression ] ;
```

*Exemples simples :*

```
type ami is (mzero,pzero,moins,plus) ;  
signal etat : ami ;  
signal temp : std_logic_vector(dimension - 1 downto 0) ;
```

Si un signal peut être connecté à plusieurs sources, ports ou drivers, une fonction de résolution doit obligatoirement lui être associée (dans la définition du sous-type), on parle alors de signal résolu (*resolved signal*)<sup>1</sup>.

L'expression optionnelle indique une valeur par défaut prise par le signal au début d'une simulation. Si cette valeur n'est pas spécifiée, le compilateur prend la valeur « de gauche » du type, la valeur minimum d'un type entier ascendant (déclaré *min to max*), par exemple.

Les signaux des catégories *register* ou *bus*, obligatoirement résolus, sont dits gardés (*guarded*), ils interviennent dans les structures de blocs que nous reverrons. Schématiquement, un signal gardé peut être connecté ou déconnecté de ses sources en fonction d'un test logique. Sa catégorie précise s'il conserve sa valeur initiale (*register*) ou prend une valeur fixée par l'utilisateur dans la fonction de résolution (*bus*) en cas de déconnexion.

Le concept de signal est plus riche qu'une simple valeur stockée (vision informatique) ou qu'un simple nœud dans un montage (vision circuits). Il s'accompagne d'informations utiles, tant en simulation qu'en synthèse, qui précisent son comportement temporel, la gestion des conflits et la synchronisation interprocessus. Lors de la synthèse, un simple signal binaire génère une équipotentielle, bien sûr, mais également la façon dont cette équipotentielle est pilotée, le fait qu'elle soit commandée par une mémoire, par un simple opérateur combinatoire ou par une porte trois états ou collecteur ouvert. La différence majeure entre les langages classiques de description de circuits programmables et VHDL réside moins dans l'aspect « Pascalien » des instructions que dans l'importance de la sémantique des signaux.

1. Voir paragraphe 2.7.3.

Terminons cette remarque sur un exemple : dans tous les langages plus ou moins dérivés de PALASM, les bascules arrivent par la syntaxe du code source, *via* des mots clés du langage ; en VHDL elles découlent de la sémantique du programme. L'une des tâches majeures de l'apprenti programmeur est d'apprendre à reconnaître immédiatement les bascules générées par son programme.

### ► Variables

Les variables jouent, dans les algorithmes séquentiels, le même rôle que leurs homonymes des langages de programmation classiques. Elles servent au stockage de valeurs nécessaires au bon déroulement d'un algorithme. Sauf pour les compteurs des boucles *for*, elles doivent toujours être déclarées, cela n'étonnera personne, dans la zone déclarative des modules séquentiels<sup>1</sup> d'un programme i.e. les processus ou les sous-programmes (fonctions ou procédures).

#### Syntaxe de déclaration :

```
variable_declaration ::=
    variable indent_list : subtype_indication [:= expression] ;
```

#### Exemples :

##### Dans une fonction :

```
function sum10(a,b : integer) return integer is
    variable tmp : integer range 0 to 18 ;
begin
    tmp := a + b ;
    if tmp > 9 then
        tmp := tmp - 10 ;
    end if ;
    return tmp ;
end sum10 ;
```

##### Dans un processus :

```
process(clear,sel)
    variable int : bit_vector(3 downto 0);
begin ...-- etc.
```

1. La norme VHDL 1993 introduit la notion de variable globale, partagée (*shared*) entre différents processus. Nous n'en parlerons pas ici pour la simple raison qu'elles sont plus une source d'ennuis qu'une facilité : elles posent des problèmes de portabilité, même en simulation, leur comportement étant contradictoire avec la double contrainte du parallélisme et de la causalité. Une variable partagée peut avoir une valeur qui dépend de l'ordre dans lequel le simulateur active les différents processus. Inutile de dire qu'en synthèse le résultat est pire.

Les signaux ne posent pas ces problèmes : vus par un informaticien, ils contiennent en eux-mêmes les « sémaphores » qui assurent la bonne synchronisation interprocessus. Pour ces questions un peu délicates nous renvoyons le lecteur à l'abondante littérature traitant des systèmes d'exploitation multitâches, dans lequel le problème se pose rigoureusement de la même façon, au chapitre communications interprocessus.



Une opération modifie immédiatement la valeur d'une variable, cela n'a rien de surprenant.

Les variables des fonctions sont dynamiques, elles ont une durée de vie nulle au sens temps simulateur. Elles sont créées à l'appel de la fonction et disparaissent lors du retour au programme appelant.

Les variables des processus ont un comportement plus pervers : elles ont une durée de vie qui va du début à la fin de la simulation, puisqu'une tâche VHDL ne meurt jamais, comme nous le verrons. Elles permettent donc de conserver une valeur au cours du temps simulateur. En synthèse cela pose un problème. En aucun cas une variable ne doit générer « quelque chose » dans le circuit, les règles de causalités entre événements ne seraient plus respectées, une variable ne peut pas avoir de temps de retard, etc. D'où d'éventuels problèmes de portabilité. Que le lecteur se rassure, le remède est simple : la bonne pratique consiste à n'utiliser les variables, dans du code synthétisable, que comme des variables dynamiques (la classe automatique du C). Pour cela il suffit de ne jamais compter sur la valeur initiale d'une variable, mais de lui donner une valeur par défaut au début du code exécutable (voir § 2.4.2 pour plus de précisions).

Les variables d'une procédure fonctionnent comme celles des fonctions, elles existent tant que la procédure est active. Mais une procédure peut contenir une instruction d'attente (`wait ...`); dans ce cas, ses variables auront une durée de vie non nulle, au sens du temps simulateur. Nous trouvons alors une situation intermédiaire entre le comportement des variables des fonctions (durée de vie nulle) et de celles des processus (éternelles).

### e) *Alias*

Il est parfois pratique de désigner par un autre nom une entité nommée qui existe déjà sous un premier nom. Cette entité peut être un objet, un sous-programme, un type, etc.

Il est important de noter qu'une déclaration d'alias ne crée rien d'autre qu'un synonyme. Elle ne crée en aucun cas un nouvel objet, la classe de l'objet visé n'est pas modifiée.

La syntaxe générale (simplifiée) de création d'un alias est :

```
alias alias_designator [: subtype_indication ] is name ;
```

#### ➤ Exemples :

```
signal instr : bit_vector(0 to 15) ;
alias opcode : bit_vector(0 to 9) is instr (0 to 9) ;
alias source : bit_vector(2 downto 0) is instr (10 to 12) ;
alias destin : bit_vector(2 downto 0) is instr (13 to 15) ;

variable mot : bit_vector(15 downto 0) ;
alias signe : bit is mot(15) ;
```

### 2.2.4 Expressions

Une expression est une formule qui définit le mode de calcul d'une valeur. Cette valeur possède un type et peut être affectée à un objet de ce type, ou être manipulée en tant que telle, de façon anonyme, dans un test par exemple.

Nous ne donnerons pas ici la forme BNF complète d'une expression. Elle est relativement longue et ne pose, somme toute, guère de problème de compréhension. Compte tenu des extensions successives du langage, nous invitons cependant le lecteur à consulter la documentation de son compilateur avant de se lancer dans l'écriture de programmes. Les opérateurs présentés ici sont ceux définis par la norme 1993.

#### a) Opérateurs

Les opérateurs sont énumérés ci-dessous par classes de priorités croissantes, de haut en bas. Des opérateurs de même classe ont donc la même priorité. Rappelons au lecteur que les opérateurs logiques `nand` et `nor` ne sont pas associatifs. Des parenthèses sont donc indispensables dans les expressions où apparaissent plusieurs de ces opérateurs.

VHDL est un langage dans lequel il est possible d'étendre le domaine d'utilisation d'un opérateur (types des opérandes) par l'écriture d'une fonction qui porte le nom de l'opérateur, c'est ce que l'on appelle la surcharge. Cette opération ne modifie pas la classe de priorité à laquelle appartient l'opérateur. Les bibliothèques spécifiques d'un outil de CAO contiennent systématiquement des opérateurs surchargés<sup>1</sup>. Les types d'opérandes évoqués ci-dessous sont ceux qui sont définis dans le langage, en l'absence de toute extension par surcharge.

<code>logical_operator</code>	<code>::=</code>	<code>and</code>	<code> </code>	<code>or</code>	<code> </code>	<code>nand</code>	<code> </code>	<code>nor</code>	<code> </code>	<code>xor</code>	<code> </code>	<code>xnor</code>
<code>relational_operator</code>	<code>::=</code>	<code>=</code>	<code> </code>	<code>/=</code>	<code> </code>	<code>&lt;</code>	<code> </code>	<code>&lt;=</code>	<code> </code>	<code>&gt;</code>	<code> </code>	<code>&gt;=</code>
<code>shift_operator</code>	<code>::=</code>	<code>sll</code>	<code> </code>	<code>srl</code>	<code> </code>	<code>sla</code>	<code> </code>	<code>sra</code>	<code> </code>	<code>rol</code>	<code> </code>	<code>ror</code>
<code>adding_operator</code>	<code>::=</code>	<code>+</code>	<code> </code>	<code>-</code>	<code> </code>	<code>&amp;</code>						
<code>sign</code>	<code>::=</code>	<code>+</code>	<code> </code>	<code>-</code>								
<code>multiplying_operator</code>	<code>::=</code>	<code>*</code>	<code> </code>	<code>/</code>	<code> </code>	<code>mod</code>	<code> </code>	<code>rem</code>				
<code>miscellaneous_operator</code>	<code>::=</code>	<code>**</code>	<code> </code>	<code>abs</code>	<code> </code>	<code>not</code>						

On notera que l'échelle de priorités n'est pas toujours évidente (mais dans quel langage est-ce le cas ?), aussi ne faut-il pas hésiter à utiliser des parenthèses en cas de doute.

1. L'exemple le plus fréquent porte sur la représentation des nombres par des vecteurs d'éléments binaires. Cette représentation est tellement fréquente qu'elle est, actuellement normalisée par la bibliothèque IEEE (voir § 2.5.3 et 2.7.4). Cela évite d'avoir recours aux bibliothèques « personnelles » de chaque fournisseur, évidemment moyennement portables.

La plupart de ces opérateurs n'appelle pas de commentaire, leur signification est identique dans la majorité des langages de programmation. Pour les autres, les opérateurs logiques par exemple, leur nom exprime leur fonction.

Nous nous contenterons ci-dessous d'apporter quelques précisions utiles :

- **Opérateurs logiques** : ce sont des opérateurs binaires (à deux opérandes) dont les opérandes et le résultat sont des scalaires ou des vecteurs (tableaux à une dimension) de type bit ou booléen. L'opérateur `xnor` (`not xor`) est nouveau (norme 1993).
- **Opérateurs relationnels** : ce sont des opérateurs binaires dont les deux opérandes sont de même type et le résultat de type booléen. L'égalité et l'inégalité acceptent des opérandes de type quelconque, sans restriction. Les relations d'ordre sont limitées aux types scalaires et aux vecteurs de types discrets. Les vecteurs sont comparés de gauche à droite.
- **Opérateurs de décalages et de rotation** : ces opérateurs, rajoutés par la norme 1993, agissent sur des vecteurs d'éléments de type bit ou boolean. Ils effectuent les quatre types de décalages classiques : logique ou arithmétique, à gauche ou à droite, et les rotations dans les deux sens. Leur opérande de gauche est le tableau, leur opérande de droite est un entier qui indique le nombre de décalages élémentaires (d'une case) à réaliser. Leur nom indique l'opération, par exemple : `sla` pour `shift left arithmetic`, `srl` pour `shift right logical`.

*Exemples :*

```
subtype octet is bit_vector(7 downto 0) ;
constant moins_77 : octet := "10110011" ;
constant moins_39 : octet := moins_77 sra 1 ;--11011001
constant plus_89  : octet := moins_77 srl 1 ;--01011001
constant plus_103 : octet := moins_77 rol 1 ;--01100111
```

- **Opérateurs additifs** : l'addition et la soustraction agissent sur deux opérandes numériques de même type. L'opérateur `&` fournit un vecteur à partir de la concaténation de deux vecteurs dont les éléments sont de même type, ou à partir d'un vecteur et d'un élément du type rajouté. La direction, ascendante ou descendante, du résultat est déterminée par l'opérande de gauche, ou à défaut par l'opérande de droite ou à défaut par une indication de sous-type explicite

*Exemples :*

```
subtype octet is bit_vector(7 downto 0) ;
constant moins_77 octet := "101" & "10011" ;
constant moins_77_bis octet := "1011001" & '1' ;
```

- **Opérateurs de signe** : ce sont des opérateurs unaires qui agissent sur n'importe quel type numérique. Leur priorité inférieure à celle des opérateurs multiplicatifs interdit des expressions comme `A/-B` qui doit être écrit `A/(-B)`.
- **Opérateurs multiplicatifs** : multiplication et division agissent sur des opérandes de même type numérique. Les opérateurs modulo et reste agissent sur des entiers.

Leurs résultats diffèrent si les opérandes sont de signes contraires :

```

7 mod 4      = 7 rem 4      = 3
-7 mod (-4)  = -7 rem (-4) = -3
7 rem (-4)   = 3
7 mod (-4)   = -1
-7 rem 4     = -3
-7 mod 4     = 1

```

Les opérateurs multiplicatifs ne sont pas toujours tous acceptés en synthèse, même pour des types entiers.

Multiplication et division sont étendues aux types physiques, sous réserve que le résultat conserve un sens physique ou soit un nombre sans dimension.

- **Opérateurs divers** : la négation logique agit sur des types bit ou boolean ou des vecteurs de ces types.

La valeur absolue agit sur n'importe quel type numérique.

L'exponentiation accepte un opérande de gauche numérique et un opérande de droite entier.

Ces deux derniers opérateurs ne sont évidemment pas synthétisables dans le cas général.

### b) Opérandes

Les opérandes d'une expression peuvent être des objets nommés, une expression entre parenthèses, un appel de fonction, une constante littérale ou un agrégat<sup>1</sup>. Dans certains cas, plutôt rares, on peut être amené à préciser ou, de façon très restrictive, modifier le sous-type d'un opérande. Les paragraphes qui suivent précisent quelques points qui n'ont pas été abordés jusqu'ici, sauf en ce qui concerne les fonctions dont nous parlerons plus loin.

#### ➤ Les noms

Les noms des objets simples, comme des constantes ou des signaux scalaires, sont formés de lettres, de chiffres et du caractère souligné ('\_'). Le premier caractère doit être une lettre.

Un membre d'un enregistrement est désigné par un nom sélectionné, traduction mot à mot de *selected name*, le terme français obtenu n'est guère évocateur. Il s'agit d'un nom composé d'un préfixe et d'un suffixe séparés par un point. Par exemple, étant donnée la déclaration :

```
signal afficheur : date ;-- type défini précédemment
```

Le champ jour du signal afficheur est repéré par le nom composé :

```
afficheur.jour
```

1. Pour être complet il faudrait rajouter les allocateurs de mémoire, mais nous ne les aborderons pas dans cette présentation.

Les noms sélectionnés interviennent également pour accéder aux objets déclarés dans des bibliothèques ; schématiquement le préfixe représente le chemin d'accès et le suffixe le nom simple de l'objet, dans une construction similaire à celle qui est utilisée pour retrouver un fichier informatique dans un système de fichiers.

L'accès aux éléments d'un tableau se fait par un nom composé qui comporte comme préfixe le nom du tableau et comme suffixe une expression, ou une liste d'expressions, entre parenthèses. Il peut se faire élément par élément (*indexed name*)<sup>1</sup> :

```
sort <= entree(sel) ;

ou, dans le cas des vecteurs uniquement, par tranche (slice name) :
```

```
mem1(5 to 12) <= mem2(0 to 7) ;
```

Le préfixe du nom d'un tableau ou d'un enregistrement réfère tous les éléments de l'objet structuré :

```
constant bastille : date := (14,jul,1789) ;
constant bast1 : date := bastille ;
```

La dernière forme de noms composés s'applique aux attributs. À un objet peuvent être rattachées des informations complémentaires que l'on appelle attributs de cet objet, nous en verrons le principe et des exemples. La référence à un attribut se fait par le nom de l'objet, en préfixe, suivi du nom de l'attribut en suffixe, séparé du précédent par une apostrophe :

```
if hor'event and hor = '1' then -- etc.

function ouex ( a : in bit_vector ) return bit is
variable parite : bit ;
begin
    parite := '0' ;
    for i in a'low to a'high loop -- etc.
```

### » Les constantes littérales

Les constantes littérales désignent des valeurs numériques, des caractères ou des chaînes de caractères, ou des chaînes de valeurs binaires. Les constantes numériques sont de type entier ou flottant et peuvent exprimer un nombre dans une base quelconque entre 2 et 16. La lettre E (ou e), pour exposant, est acceptée tant pour les types entiers que pour les types flottants. La base par défaut est la base 10.

#### *Des nombres :*

```
14      0      1E4    123_456 -- des entiers en base 10
14.0    0.0    1.0e4  123.456 -- des flottants en base 10
2#1010# 16#A#  8#12#  -- l'entier 10
16#F.FF#E+2 2#1.1111_1111_111#E11 -- le flottant 4095.0
```

1. On peut remarquer que la syntaxe peut conduire à une ambiguïté avec celle qui correspond à un appel de fonction. C'est au programmeur de veiller à ce que le contexte de l'instruction lève toute ambiguïté.

*Des caractères et des chaînes :*

```
'a'   'B'   '5'           -- des caractères
"ceci est une chaîne" "B"  -- des chaînes de
                             -- caractères
```

*Des chaînes binaires :*

```
B"1111_1111"  -- équivaut à "11111111"
X"FF"         -- la même chaîne
0"377"        -- équivaut à "01111111"
```

La différence entre chaînes binaires et chaînes de caractères équivalentes réside dans les facilités d'écritures apportées par la base et les caractères soulignés autorisés dans les premières pour faciliter la lecture. Il est important de noter que le nombre de caractères binaires générés dépend de la base, les deux derniers exemples ne sont donc pas équivalents, même si leurs équivalents numériques le sont.

## ► Les agrégats

Les agrégats constituent une notation efficace pour combiner plusieurs valeurs de façon à constituer une valeur de type structuré (enregistrement ou tableau). Compte tenu de leur utilisation fréquente, nous allons un peu détailler la syntaxe de leur construction.

```
aggregate ::=
  ( element_association { , element_associaton } )
element_association ::=
  [ choices => ] expression
choices ::=
  choice { | choice }
choice ::=
  simple_expression
  | discrete_range
  | element_simple_name
  | others
```

Un agrégat est construit comme une liste d'associations élémentaires, qui fait correspondre une valeur à un ou plusieurs membres d'un type structuré. Le type doit évidemment être déductible du contexte de l'expression. Chaque association élémentaire peut se faire par position - on ne précise pas explicitement le membre destinataire, les valeurs sont énumérées dans l'ordre utilisé dans la définition du type - ou en spécifiant le ou les membres destinataires explicitement. Dans ce dernier cas, il est possible de créer des associations « collectives », qui sont pratiques pour initialiser plusieurs éléments d'un tableau à une même valeur, par exemple.

*Exemples :*

```
-- associations par position
type table_verite_2 is array(bit,bit) of bit ;
constant ouex_tv  : table_verite_2 := (('0','1'),('1','0')) ;
constant ouex_tv1 : table_verite_2 := ("01","10") ;
constant bastille : date := (14,jul,1789) ;
```

```

-- associations explicites
constant bast2 : date := (mois => jul, annee => 1789, jour => 14) ;
constant ouex_tv2 : table_verite_2 := ('0'=>('0'=>'0', '1'=>'1'),
                                         '1'=>('0'=>'1', '1'=>'0'));
-- tableau à deux dimensions ::= vecteur(vecteur).
-- associations collectives
constant rom_presque_vide : mem8 := (0 to 10 => 3, others => 255);
constant rom_vide : mem8 := (others => 16#FF#) ;
donnee <= temp when direction = '0' else (others => 'Z') ;

```

Si on panache les différents modes d'association, les associations par position doivent être mises en premier, celles par référence en dernier. Le mot clé *others*, s'il est utilisé, ne peut que définir la dernière association de la liste.

### ➤ Précisions de types

VHDL ne tolère *a priori* pas les mélanges de types. Deux cas particuliers, de natures fort différentes, il est vrai, conduisent parfois (rarement, de fait) un programmeur à préciser le type d'une construction :

- Le résultat de l'évaluation d'une expression ou d'un agrégat peut être d'un type ambigu. On peut alors préciser ce type par une expression qualifiée :

```

type x01z is ('x','0','1','z') ;
constant zero : bit := '0' ;-- type bit
constant zerol : x01z := x01z('0') ;-- expression qualifiée

```

Rarement nécessaire, cette précision peut être utile, par exemple, quand des fonctions ou des procédures existent sous plusieurs formes qui dépendent des types des opérandes (surcharge).

- Entre deux types « proches » (*closely related*) il est possible de forcer une conversion de type. L'exemple le plus classique est celui des types numériques entre lesquels les conversions sont permises :

```

constant pi : real := 3.14159 ;
constant intpi : integer := integer(10000.0*pi)

```

Le seul autre cas de conversions autorisées par le langage concerne des tableaux qui ont le même nombre de dimensions, et dont les indices et les éléments sont de mêmes types.

## 2.2.5 Attributs

Comment détecter le changement d'un signal ? Comment un sous-programme fait-il pour accéder à la plage de variation et à la direction, ascendante ou descendante, des indices d'un tableau passé en argument ? Comment indiquer à un outil de synthèse le brochage souhaité pour un circuit ?

Toutes ces informations complémentaires, attachées aux objets<sup>1</sup> nommés qui

1. Objet est pris ici dans un sens plus large que celui vu à propos des déclarations. Le manuel de référence emploie le terme *named entities*, entités nommées. Nous n'avons pas conservé le mot entité de peur d'une confusion avec l'entité du couple entité-architecture.

forment un programme, sont le domaine des attributs. Dans le langage on dit que les attributs *décorent* les objets auxquels ils sont attachés. Les attributs se divisent en deux grandes catégories : les attributs prédéfinis et les attributs définis par l'utilisateur. Les premiers peuvent avoir un comportement dynamique, changer de valeur au cours d'une simulation, par exemple. Certains d'entre eux sont des fonctions que l'on appelle en leur passant un argument. Les seconds sont toujours des constantes.

La désignation d'un attribut a été vue précédemment à propos des noms composés. Rappelons que la référence à un attribut fait intervenir un préfixe et un suffixe séparés par une apostrophe comme dans l'expression

```
if hor'event and hor = '1' then -- etc.
```

dans laquelle le préfixe *hor* désigne un signal déclaré par ailleurs, et le suffixe *event* un attribut qui décore ce signal et renvoie la valeur booléenne vraie si le signal vient de changer de valeur.

### a) Attributs prédéfinis

Les attributs prédéfinis sont classés suivant la nature de leur préfixe et la catégorie de l'attribut lui-même : valeur, type, domaine de valeurs, fonction ou signal.

Donnons quelques points de repères avant de rentrer plus dans les détails :

- Les attributs qui se rapportent à un type scalaire, à un objet ou à un type tableau permettent essentiellement d'écrire des sous-programmes généraux. Il est possible, par exemple, de créer une fonction qui effectue une opération arithmétique sur des vecteurs de bits interprétés comme la représentation en base 2 d'un nombre. L'utilisation des attributs permet de s'affranchir des bornes et de la direction des indices des arguments de la fonction :

```
type UNSIGNED is array (NATURAL range <> ) of BIT;
--...
function "+" (L, R: UNSIGNED) return UNSIGNED is
  constant L_LEFT: INTEGER := L'LENGTH-1;
  constant R_LEFT: INTEGER := R'LENGTH-1;
  constant SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
begin -- etc .
```

La fonction "+", qui redéfinit l'addition, fonctionne quelles que soient les dimensions réelles des vecteurs qui lui sont passés en argument (l'extrait est tiré de la librairie IEEE, la mise en relief avec des caractères gras est un ajout des auteurs).

- Les attributs qui se rapportent à des signaux servent essentiellement à tester les conditions dans lesquelles un événement vient de se produire. En synthèse, le simple test d'un événement (*if hor'event ...*) revient à créer un opérateur sensible aux fronts; en modélisation, des attributs plus élaborés permettent de contrôler le respect de spécifications dynamiques. L'exemple non synthétisable suivant modélise une bascule D avec contrôle du respect des temps de set-up et hold :

```
library ieee ;
use ieee.std_logic_1164.all ;
```



```

entity bascd is
  generic ( tp : time := 3 ns ; -- retard
            tsu : time := 8 ns ; -- prépositionnement, > 0
            tho : time := 2 ns ) ; -- maintien > 0
  port ( hor,d : in std_logic ;
        q : out std_logic ) ;
end bascd ;

architecture test of bascd is
begin
  process
  begin
    wait on hor,d ;
    if d'event and hor = '1' and not hor'stable(tho) then
      report bascd'path_name & " : violation de thold" ;
      q <= 'X' after tp ;
    elsif
      hor'event and hor'last_value = '0' and hor = '1' then
        if d'last_event < tsu then
          report bascd'path_name & " : violation de set up" ;
          q <= 'X' after tp ;
        else
          q <= d after tp ;
        end if ;
      elsif
        hor'event and (hor = '1' or hor'last_value = '0') then
          q <= 'X' after tp ;
          report bascd'path_name &
            " : transition d'horloge illégale" ;
        end if ;
      end process ;
  end test ;

  message affiché lors de la simulation en cas de faute:
  ** Note: :bascd: : violation de set up

```

- Les attributs qui se rapportent aux noms des choses permettent de retrouver le chemin d'accès dans une construction hiérarchique. Ils n'ont évidemment d'utilité qu'en modélisation.

Les tableaux qui suivent résument les rôles des attributs définis par le langage. Nous les avons classés suivant la catégorie de leur préfixe, c'est-à-dire suivant la nature des objets qu'ils décorent<sup>1</sup>.

1. Le nombre des attributs prédéfinis est considérable. Qui trop embrasse mal étreint, quand on explore les modèles de simulations fournis par les constructeurs, on se rend compte que seuls les plus classiques sont utilisés couramment : domaines de définition des indices de tableaux, test d'un événement, valeur précédente d'un signal.

Attributs qui décorent un type scalaire	
attribut	valeur retournée
T'base	renvoie le type de base d'un type dérivé, sert de préfixe
T'left	borne de gauche du type
T'right	borne de droite du type
T'high	borne supérieure du type
T'low	borne inférieure du type
T'ascending	TRUE si le type T est ascendant, FALSE autrement
T'image(expr.)	renvoie une chaîne de caractères image de l'expression
T'value(chaîne)	calcule de l'expression dont la chaîne est l'image
T'pos(v)	position de l'élément v dans le type T, discret ou physique
T'val(p)	valeur de l'élément de position p dans T, discret ou physique
T'succ(v)	valeur qui suit (position + 1) l'élément de valeur v dans le type
T'pred(v)	valeur qui précède (pos. - 1) l'élément de valeur v dans le type
T'leftof(v)	valeur de l'élément juste à gauche de l'élément de valeur v
T'rightof(v)	valeur de l'élément juste à droite de l'élément de valeur v

Attributs qui décorent un tableau ou un type tableau	
attribut	valeur retournée
A'left(n)	borne de gauche de l'indice de la dimension n, n=1 par défaut
A'right(n)	borne de droite de l'indice de la dimension n, n=1 par défaut
A'high(n)	borne maximum de l'indice de la dimension n, n=1 par défaut
A'low(n)	borne minimum de l'indice de la dimension n, n=1 par défaut
A'range(n)	plage de variation de l'indice de la dimension n, n=1 par défaut
A'reverse_range(n)	plage de variation, retournée (to ↔ downto), de l'indice de la dimension n, n=1 par défaut
A'length(n)	nombre d'éléments de la dimension n, n=1 par défaut
A'ascending(n)	TRUE si l'indice de la dimension n est ascendant, FALSE autrement

Attributs qui décorent un signal	
attribut	valeur retournée
S'event	valeur TRUE si un événement vient d'affecter S, si non FALSE
S'last_event	type time, temps écoulé depuis le dernier événement sur S
S'last_value	valeur de S avant son dernier changement
S'delayed(T)	retourne un signal toujours égal à S retardé du temps constant T
S'stable(T)	signal égal à TRUE si S n'a pas subi d'événement depuis T
S'quiet(T)	signal égal à TRUE si S n'a pas été actif depuis T <sup>a</sup>
S'transaction	signal de type bit qui change d'état à chaque transaction sur S
S'active	valeur TRUE si S est actif au temps actuel
S'last_active	type time, temps écoulé depuis la dernière activité de S
S'driving	FALSE si la transaction dans le process en cours est <i>null</i>
S'driving_value	valeur de S pour la transaction dans le process en cours

- a. La différence entre activité et événement est subtile, la première est relative au pilote du signal, le second indique un changement de valeur du signal, donc visible de l'extérieur. Le pilote peut être actif et fournir une nouvelle valeur du signal identique à la précédente, donc ne pas créer d'événement.

Attributs qui décorent un objet (au sens large) nommé	
attribut	valeur retournée
E'simple_name	chaîne de caractère qui contient le nom de l'objet nommé
E'instance_name	chaîne de caractère qui contient le nom complet de l'objet nommé, avec son chemin d'accès dans la hiérarchie
E'path_name	chaîne de caractère qui contient le nom simple de l'objet nommé, avec son chemin d'accès dans la hiérarchie

Certains des attributs qui décorent les signaux n'ont aucun sens dans le cadre de la synthèse de circuits. Nous les avons, de préférence, mis à la fin de la liste<sup>1</sup>. Il en va de même pour les attributs du dernier tableau qui sont essentiellement une aide au diagnostic lors de la simulation.

1. On pourrait même penser que pour les signaux seul l'attribut 'event a un sens. Les suivants interviennent cependant dans la librairie IEEE, utilisée en simulation et en synthèse, pour des fonctions de manipulation du type `std_logic`. De toute façon les outils de synthèses n'acceptent souvent qu'un sous-ensemble du langage; il est plus que conseillé de consulter leur documentation en cas de doute.

### b) Attributs définis par l'utilisateur

L'utilisateur peut définir ses propres attributs, sous réserve qu'ils correspondent à des constantes. La plupart des fournisseurs de logiciels de synthèse utilisent le mécanisme des attributs pour permettre au concepteur de transmettre des directives à l'optimiseur et à l'outil de placement routage. Par exemple, les attributs permettent d'associer un numéro de broche, dans le circuit cible, aux ports d'entrée-sorties de l'entité supérieure d'une hiérarchie.

#### ➤ Syntaxe

Comme tout objet un attribut se déclare :

```
attribute_declaration ::=
    attribute identifier : type_mark ;
```

Et une valeur lui est associée par une *spécification d'attribut* :

```
attribute_specification ::=
    attribute identifier of entity_specification is expr ;
entity_specification ::=
    entity_name_list : entity_class ;
```

Où le mot `entity_class` désigne la catégorie de l'objet décoré : entité, signal, fonction, etc.

#### ➤ Exemples

Pour fixer le brochage d'un composant, par exemple, on trouve

```
attribute pinnum : string; -- Must define the attribute
attribute pinnum of Clk : signal is "1";
attribute pinnum of Clr : signal is "2";
```

pour le logiciel SYNARIO, et

```
ATTRIBUTE pin_numbers of and5Gate:ENTITY IS
    "a(0):2 a(1):3 " --The spaces after 3 and 5 are necessary
    & "a(2):4 a(3):5 " --for concatenation (& operator)
    & "f:6";
```

pour le logiciel WARP. Inutile de préciser qu'à ce niveau la portabilité n'est (malheureusement) plus de mise.

## 2.3 PARALLÉLISME ET ALGORITHMES SÉQUENTIELS

Commençons par une image. On peut définir un opérateur `//` qui, étant donné deux nombres réels  $x$  et  $y$  leur associe un nombre  $z$  défini par :

$$z = x // y = \frac{x * y}{(x + y)}$$

Si on demande à un mathématicien et à un électronicien de démontrer que l'expression  $a // b // c$  a un sens, ils auront probablement deux approches très différentes.

Le mathématicien démontrera que // est commutatif, ce qui est immédiat, et associatif, ce qui l'est un peu moins. Puis il appliquera un théorème bien connu de la théorie élémentaire des ensembles ; C.Q.F.D.

L'électronicien remarquera que // définit la valeur de deux résistances associées en parallèle, et qu'il est évident que pour en associer trois on peut commencer par en souder deux, puis la troisième, dans un ordre quelconque.

L'histoire de la concurrence, dans un langage comme VHDL, présente certaines similitudes avec notre histoire.

On peut l'aborder d'un point de vue informatique, il faudra alors s'attacher à bien comprendre la notion de concurrence entre tâches (ou processus), qui apporte dans ses bagages celle de communications et de synchronisations entre tâches. Les signaux jouent, dans ce contexte, le rôle d'outils d'échanges qui contiennent, nous l'avons évoqué précédemment, la notion de sémaphore. La nature même d'une action sur un sémaphore est qu'elle est indivisible : toute action commencée se déroule jusqu'à son terme sans qu'une autre tâche puisse modifier cette valeur prématurément. C'est une condition sine qua non de l'intégrité de tout système. La traduction en VHDL de ce principe est que *dans* un processus, les actions sur un signal sont indivisibles, seul a un sens le résultat final de l'algorithme qui ne sera visible que quand le processus se mettra en sommeil, dans l'attente d'un événement. En réalité le processus ne modifie pas le signal, il transmet au simulateur<sup>1</sup>, une proposition de nouvelle valeur (une *transaction*), pour un instant futur ou pour le même instant (*projected waveform*) ; le noyau seul décide de la valeur réellement prise par le signal, quand le processus lui rend la main. Ce qui se passe *pendant* l'activité d'un processus n'a pas d'autre réalité que la triste contrainte du temps de calcul fini de nos ordinateurs. Mesuré en temps simulateur, virtuel, ce *pendant* a une durée nulle.

On peut également aborder la question de la concurrence d'un point de vue circuits. À un niveau élémentaire, on retrouve les résistances en parallèle : tous les opérateurs d'un circuit coexistent et interagissent simultanément, quel que soit l'ordre dans lequel on décrit leurs opérations et interconnexions. À un niveau un peu moins élémentaire, il est intéressant de réfléchir à la modélisation des opérateurs synchrones (*edge triggered flip flops*) idéaux, c'est-à-dire dépourvus de temps de propagations parasites qui limitent leurs performances. Les fronts d'horloge séparent l'*avant* de l'*après*. L'état du système après le front est une fonction de la situation qui prévalait avant ce front. La question de ce qui se passe *pendant* n'est pas du ressort de la logique, c'est l'affaire de l'analogicien qui conçoit le circuit au niveau des transistors. Nous retrouvons le même temps virtuellement nul pour le *pendant*. On peut étendre cette vision du temps à n'importe quel opérateur logique, mais alors la séparation entre avant et après est faite par un événement, un changement, qui porte sur une entrée quelconque de l'opérateur. Il apparaît immédiatement une difficulté conceptuelle, qui correspond à une difficulté réelle des fonctionnements

---

1. Le simulateur joue, dans ce mécanisme, le même rôle qu'un noyau de système d'exploitation multitâches. Le terme de noyau (*kernel*) est d'ailleurs courant dans les documents qui concernent VHDL.

asynchrones : comment traiter plusieurs événements simultanés ? Il faut donc à chaque instant traiter non pas un événement, mais une liste d'événements. Le traitement d'un événement peut en créer un autre, éventuellement au même instant. Le temps n'a le droit d'évoluer que quand tous les événements de l'instant ont été traités. Rien ne garanti que le processus converge, c'est-à-dire fasse évoluer le temps ... virtuel du simulateur, bien sûr.

La bonne compréhension du sens des choses passe, à notre point de vue, par un aller retour permanent entre ces deux aspects, logiciel et circuits. Il est vrai que le premier est un peu plus abstrait que le second. En cas de doute le recours est toujours de se demander si une description a un sens quand on essaye d'imaginer le circuit qui se cache derrière.

### 2.3.1 Le corps d'une architecture : le monde concurrent

Les instructions qui décrivent une architecture sont des processus indépendants, qui communiquent entre eux par les signaux du schéma. Dans les cas simples ces processus sont implicites, l'utilisateur les crée en écrivant des instructions concurrentes, ce sont elles que nous allons aborder dans un premier temps.

#### a) Affectations de signaux

L'instruction concurrente la plus élémentaire, du point de vue syntaxique, est l'affectation d'une valeur à un signal. Le symbole réservé à l'affectation à un signal est `<=`. Un simple opérateur logique ou s'écrit, par exemple :

```
sort <= sele0 or sele1 ;
```

#### ➤ Syntaxe

L'affectation concurrente offre plus de souplesse que le simple transfert de la valeur d'une expression à un signal. Il est possible d'y inclure des tests par une affectation conditionnelle :

```
plusout <= '1' when etat = plus else '0' ;
```

ou par une affectation sélective :

```
with sel4 select
    sort <= e0 when zero ,
        e1 when un ,
        e2 when deux ,
        e3 when trois ,
        '0' when others ;
```

Le mot réservé `others` regroupe toutes les combinaisons non testées du signal de sélection.

La syntaxe complète de l'affectation concurrente est encore plus générale, mais ne prend tout son sens que dans un contexte de simulation. La valeur affectée à la cible (*target*) peut être une forme d'onde (*waveform*), qui est une liste de couples valeur-temps. Une étiquette (*label*), optionnelle, peut être rajoutée pour nommer

l'instruction (cette règle est générale) et des options peuvent préciser le comportement temporel du signal (en simulation) et une garde :

```

concurrent_signal_assignment ::=
    [ label :] conditional_signal_assignment
    | [ label :] selected_signal_assignment
conditional_signal_assignment ::=
    target <= options { waveform when condition else }
    waveform [when condition];
selected_signal_assignment ::=
    with expression select
    target <= options { waveform when choices , }
    waveform when choices;
waveform ::=
    waveform_element {, waveform_element}
    | unaffected
waveform_element ::=
    value_expression [ after time_expression ]
    | null [ after time_expression ]
options ::=
    [guarded] [transport | [reject time_expression] inertial]
choices ::= choice { | choice }
choice ::= static_expression | discrete_range | others
target ::= name | aggregate

```

Nous avons donné ici une description syntaxique relativement complète<sup>1</sup> de l'affectation concurrente, en raison de son utilisation très fréquente. Nous ne pouvons que conseiller au lecteur de se familiariser progressivement avec les multiples constructions possibles, en gardant à l'esprit que la simplicité et la lisibilité sont des garants de bonne santé d'un programme.

Le mot clé `unaffected` indique que le pilote du signal ne doit pas être activé, le signal conserve donc sa valeur précédente.

Les mots clés `guarded` et `null` s'adressent à des signaux gardés dont nous reparlerons à propos des blocs.

Les mots clés `transport`, `inertial` et `reject` précisent la façon dont le simulateur doit traiter des impulsions (parasites) de courtes durées. Un paragraphe est consacré à ce sujet.

La cible (`target`) peut être un nom de signal ou, mais cela est d'une utilisation rare, un agrégat comme dans l'exemple ci-dessous :

```

entity divers is
    port (a,b,c : out integer );
end divers ;
architecture test of divers is
    type table is array(0 to 2) of integer ;

```

1. La norme 1993 a rajouté une option globale de comportement du processus sous-jacent : `posponed`. Cette option assure que le processus ne sera activé qu'au dernier cycle de simulation du temps courant, quand tous les signaux sont stabilisés.

```

begin
    (a,b,c) <= table'(others => 3)
end test ;

```

Il est important de comprendre que le parallélisme implique une permanence de l'instruction. Du point de vue informatique, le processus sous-jacent est activé à chaque événement susceptible de modifier la valeur du signal destinataire de l'affectation. Du point de vue circuit, l'affectation concurrente correspond à des connexions fixes entre des opérateurs logiques.

### ► Instant présent et futur : les événements

Revenons sur le temps, pas celui qui passe pour le concepteur, celui du circuit modélisé, celui qui se mesure en temps simulateur, dont la valeur actuelle s'appelle *now* dans le langage.

Prenons pour support un exemple simple, celui de la figure 2-5.

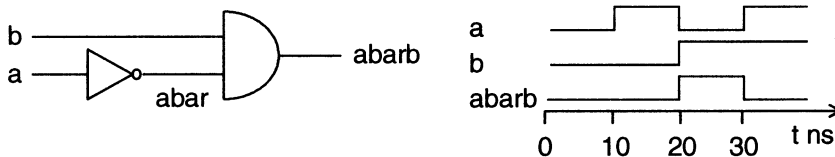


Figure 2-5 Temps et événements

Le programme VHDL correspondant est le suivant :

```

entity op_abarb is
    port ( a, b : in bit ;
          abarb : out bit ) ;
end op_abarb ;

architecture concurrente of op_abarb is
    signal abar : bit ;
begin
    abarb <= abar and b ;
    abar <= not a ;
end concurrente ;

```

Suite à une simulation, il est possible d'obtenir du simulateur, outre des valeurs en fonction du temps, la liste des événements qui affectent les signaux du montage :

ns	delta	a	b	abarb	abar
0	+0	0	0	0	0
0	+1	0	0	0	1
10	+0	1	0	0	1
10	+1	1	0	0	0
20	+0	0	1	0	0
20	+1	0	1	0	1
20	+2	0	1	1	1
30	+0	1	1	1	1
30	+1	1	1	1	0
30	+2	1	1	0	0



Par défaut la valeur initiale de tous les signaux est '0' (bit'left). Un cycle de simulation place abar à '1', mais laisse la sortie abarb à '0'. L'événement suivant survient à  $t = 10$  ns, provoqué par le changement de valeur de l'entrée a. Cet événement en provoque un second, qui affecte abar, au même instant, mais qui en est la conséquence. À  $t = 20$  ns, l'entrée b passe à '1', provoquant une cascade d'événements simultanés, mais dont le traitement correct, pour respecter les chaînes de causalité, nécessite trois cycles de simulation à cet instant. Etc.

Chaque cycle de simulation est appelé un cycle *delta*. De delta en delta le simulateur évolue de la situation actuelle vers la situation future. On peut, pour faciliter la compréhension, imaginer que delta représente un temps infinitésimal, qui sert à assurer le traitement correct de la causalité, en introduisant une sorte de chronologie microscopique. À notre sens, cette assimilation de delta à un temps n'est pas nécessaire, voire troublante : quel que soit le nombre de cycles delta, leur accumulation correspond à un temps toujours nul.

Rien n'assure que le processus de calcul de la situation future converge. Modifions légèrement notre schéma comme indiqué figure 2-6.

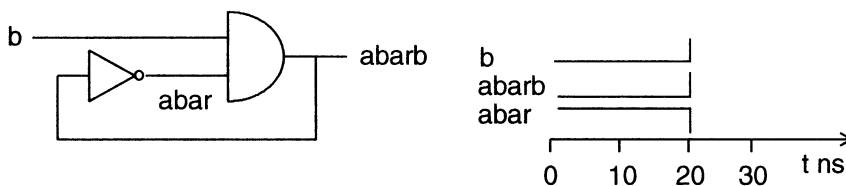


Figure 2-6 Un montage où le temps s'arrête.

Le programme correspondant est des plus simples :

```
entity enfer is
  port ( b : in bit ;
        abarb : buffer bit ) ;
end enfer ;

architecture concurrente of enfer is
  signal abar : bit ;
begin
  abarb <= abar and b ;
  abar <= not abarb ;
end concurrente ;
```

Lors de la simulation de ce programme, avec le même stimulus pour l'entrée b que précédemment, c'est-à-dire '0' jusqu'à 20 ns, '1' ensuite, le temps semble s'arrêter à 20 ns. Heureusement, le simulateur nous avertit par un message d'erreur :

```
# Iteration limit reached. Increase limit to continue.
```

Un coup d'œil à la liste des événements montre que le résultat oscille entre les deux valeurs binaires possibles, indéfiniment, mais en un temps nul :

ns	delta	b	abarb	abar
0	+0	0	0	0
0	+1	0	0	1
20	+0	1	0	1
20	+1	1	1	1
20	+2	1	1	0
20	+3	1	0	0
20	+4	1	0	1
20	+5	1	1	1
20	+6	1	1	0
20	+7	1	0	0
20	+8	1	0	1
etc.				

Il s'agit de simulation logique ; qu'en est-il de la réalité ? Tout électronicien connaît la réponse, si les opérateurs ont suffisamment de gain et un retard non nul, le résultat est réellement oscillatoire, ce qui se simule évidemment sans problème. Si les conditions de Nyquist sont telles que le montage n'est pas oscillatoire, la sortie se stabilise à un niveau analogique, entre le niveau haut et le niveau bas, en quelque sorte à la valeur moyenne des hésitations du simulateur.

Soulignons, pour finir cette discussion, une tautologie : entre deux événements aucun signal ne change de valeur. C'est la définition même des événements. Cette façon d'aborder le fonctionnement des circuits conduit à un langage qui privilégie le concept de mémoire : un oubli est en fait l'oubli d'un changement, donc un maintien dans l'état.

#### ► Causalité et parallélisme : les pilotes

Revenons sur l'espace. Dans l'architecture des programmes précédents, le fonctionnement ne dépend pas de l'ordre dans lequel on écrit les deux instructions, le contraire serait choquant. Nous les avons écrites volontairement de la sortie vers l'entrée pour bien indiquer qu'il n'y a pas de lien entre l'ordre d'écriture et une quelconque chronologie. Quand on écrit

```
abarb <= abar and b ;
abar  <= not a ;
```

on ne présuppose pas un ordre d'exécution, ce qui serait en l'occurrence absurde. On indique simplement que le signal `abarb` est sensible aux événements qui affectent `abar` et `b` ; et que le signal `abar` est sensible aux événements qui affectent `a`. On décrit simultanément les expressions qui calculent les nouvelles valeurs des signaux. Ces expressions sont réévaluées chaque fois qu'un événement affecte l'un de leurs opérandes. Cela revient à établir ce que certains appellent une connexion permanente, mais le mécanisme profond est plus subtil qu'une simple soudure entre des fils. Nous verrons que les processus explicites, l'instruction `process`, permettent de dissocier les signaux d'activation et les opérandes des équations.

En simulation chaque instruction se comporte comme un processus indépendant des autres, qui communique avec le noyau au moyen des signaux :

- Pour chaque processus, explicite ou implicite, le noyau tient à jour la liste des événements susceptibles de le réveiller,
- Pour chaque signal, objet d'une affectation, le noyau tient à jour un pilote qui indique les formes d'ondes souhaitées. Le signal lui même est mis à jour par le noyau en fonction du ou des pilotes qui lui sont attachés. Un processus ne modifie pas directement la valeur d'un signal, il introduit une nouvelle valeur dans un pilote.

La figure 2-7 tente d'illustrer ce mécanisme.

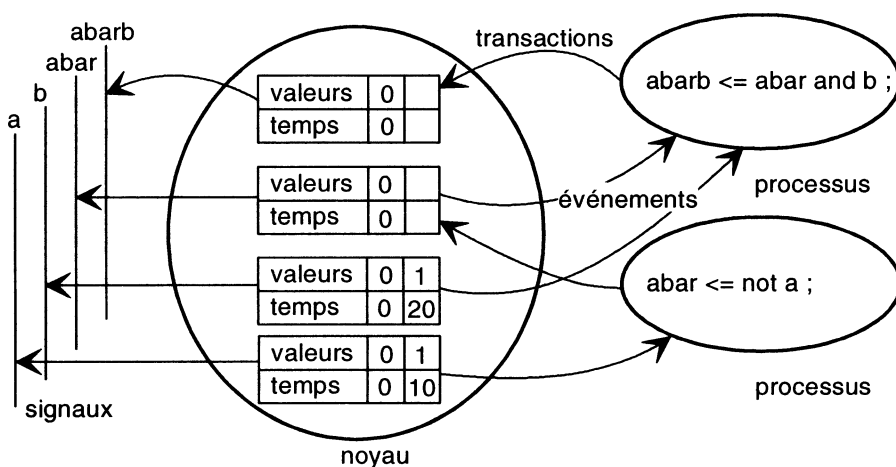


Figure 2-7 Signaux, pilotes et noyau.

Sur cette figure les entrées apparaissent comme des signaux ordinaires, cette simplification ne change rien au principe qui est discuté ici (il manque le processus qui génère les formes d'ondes de a et b). Le noyau signale aux processus les événements qui les concernent, les processus retournent au noyau des transactions ; d'où les sens des flèches.

#### » Unicité de la valeur d'un signal et conflits

Le parallélisme peut entraîner des conflits. Rien de plus normal, si plusieurs pilotes tentent de fixer des valeurs différentes à un signal il y a un problème ; il en est de même dans la réalité, le câblage en parallèle de plusieurs sorties d'opérateurs ne peut se faire sans précaution. Dans la majorité des cas il s'agit d'une erreur, dans le monde réel comme dans le monde virtuel.

Pourtant les structures de bus et de ou câblés existent. Elles s'appliquent à des opérateurs particuliers, dont les sorties ne sont pas des simples sources de tensions binaires. En VHDL ces signaux sont déclarés de façon particulière, leurs types sont accompagnés d'une fonction de résolution des conflits. Nous aurons l'occasion d'y revenir.

Retenons que sauf précaution particulière, un signal ne doit avoir qu'un seul pilote, c'est-à-dire qu'il ne peut pas faire l'objet de plusieurs affectations concurrentes.

### ➤ Opérateurs logiques combinatoires

Les affectations concurrentes sont idéales pour modéliser les opérateurs combinatoires simples, elles reflètent directement la structure des équations d'un schéma. L'exemple donné pour l'affectation sélective, un multiplexeur à quatre entrées, en est une illustration.

Rien n'empêche, *a priori*, de les utiliser pour décrire des opérateurs séquentiels élémentaires, mais la manœuvre est souvent dangereuse. Le parallélisme est difficile à maîtriser quand interviennent des rétrocouplages (actions des sorties sur les opérandes), l'exemple précédent du temps qui s'arrête n'est qu'un pâle aperçu de ce qui attend le programmeur imprudent<sup>1</sup>.

### b) Instanciation de composants

Une fonction complexe est généralement construite de façon hiérarchique : assemblages de blocs plus simples interconnectés par des signaux, dans une description structurelle. Chaque bloc lui-même peut être le sommet d'une nouvelle hiérarchie, et ainsi de suite, jusqu'à arriver à des structures suffisamment simples pour être décrites directement, ou être des primitives d'une librairie de composants élémentaires.

Vu d'un niveau hiérarchique, un sous-élément est un composant ; ce composant peut lui-même être décrit comme une unité de conception (un couple entité-architecture). L'opération d'*instanciation* consiste à établir les liens entre les signaux d'un niveau et les ports d'accès du niveau inférieur et, le cas échéant, à fixer les valeurs de paramètres génériques. Nous nous contenterons ici d'un premier aperçu de l'opération, que nous étudierons plus complètement un peu plus loin. Le cas simple envisagé ici suppose que les noms des composants et de leurs ports d'accès soient identiques à ceux de l'entité, déjà compilée, disponible dans l'environnement de travail courant désigné symboliquement par *work*.

### ➤ Syntaxe

Pour utiliser un composant il faut le déclarer, dans la zone de déclaration de l'architecture ou dans un paquetage visible depuis cette architecture, et l'instancier dans la zone des instructions.

La déclaration reproduit à peu de choses près celle, dans sa forme minimum, d'une entité ; ce qui n'a rien d'étonnant :

```

component component_name [ is ]
    [ generic (generic_list) ; ]
    [ port (port_list) ; ]
end component [ component_name ] ;

```

1. Dans la réalité les rétrocouplages asynchrones ne sont pas plus faciles à dominer ! Ils sont à l'origine d'aléas divers et variés, les simulateurs ne font que refléter ici le comportement de schémas dont les règles de conception sont malsaines.

L'utilisation du composant se fait par une instruction d'association précédée d'une étiquette obligatoire<sup>1</sup> :

```
label : component_name [ generic map (association_list) ]
      [ port map (association_list) ] ;
```

Les listes d'associations établissent la correspondance entre les noms internes au composant et les noms des signaux correspondants, ou les expressions constantes dans le cas des paramètres génériques. Le mécanisme d'association peut être implicite, par position, ou explicite, dans une syntaxe proche de celle rencontrée à propos des agrégats :

```
association_list ::=
    association_element{, association_element }
association_element ::= [formal_part =>] actual_part
```

Les paramètres formels désignent les noms internes au composant, les paramètres réels les grandeurs visibles dans l'architecture qui réalise l'instanciation.

### » Exemple

Un exemple d'association implicite, par position, a été présenté à propos de la description structurée d'un multiplexeur au moyen de portes élémentaires, au paragraphe 2.2.2.

L'exemple qui suit a été obtenu automatiquement à partir de la représentation schématique (figure 2-8) d'un compteur décimal à deux décades<sup>2</sup>.

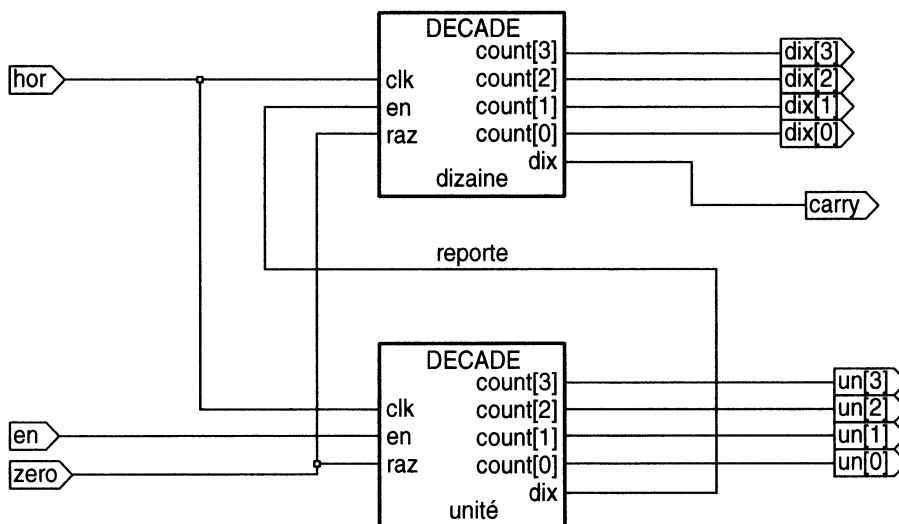


Figure 2-8 Structure d'un compteur décimal à deux chiffres.

1. La norme VHDL 93 permet également d'instancier directement une entité, sans déclaration de composant. Il faut alors faire précéder le nom du mot clé `entity`.
2. Logiciel SYNARIO de DATA I/O.

Ce type de présentation graphique facilite la lisibilité d'une documentation, l'instruction `port map` n'étant pas toujours très visuelle.

Le programme généré utilise des associations explicites :

```
entity CNTDEC is
  Port ( en : In      std_logic;
        hor : In      std_logic;
        zero : In     std_logic;
        carry : Out   std_logic;
        dix : Out     std_logic_vector (3 downto 0);
        un : Out      std_logic_vector (3 downto 0) );
end CNTDEC;
architecture SCHEMATIC of CNTDEC is

  signal  reporte : std_logic;

  component DECADE
    Port ( clk : In      std_logic;
          en : In      std_logic;
          raz : In      std_logic;
          count : Out   std_logic_vector (3 downto 0);
          dix : Out     std_logic );
  end component;

begin
  unite : DECADE Port Map ( clk=>hor, en=>en, raz=>zero,
    count(0)=>un(0), count(1)=>un(1), count(2)=>un(2),
    count(3)=>un(3), dix=>reporte );
  dizaine : DECADE Port Map ( clk=>hor, en=>reporte,
    raz=>zero, count(0)=>dix(0), count(1)=>dix(1),
    count(2)=>dix(2), count(3)=>dix(3), dix=>carry );
end SCHEMATIC;
```

Si l'instruction `port map` apparaît comme un peu rébarbative, il faut bien l'avouer, elle est cependant infiniment plus souple qu'une représentation schématique. Nous verrons, en particulier, qu'il est possible de créer des motifs répétitifs par des boucles qui calculent les paramètres d'association.

### c) Modules de programme

Une description complexe peut être subdivisée en composants indépendants, nous venons d'en avoir un aperçu. D'autres méthodes, complémentaires de la précédente, existent pour diviser un problème afin de le résoudre. Nous examinons ici des divisions qui reflètent directement des structures matérielles dans les circuits : les blocs et les processus. Anticipons un peu en signalant qu'il existe des outils purement logiciels, parfaitement utilisables en synthèse : les fonctions et les procédures, dont nous reparlerons.

Les blocs et les processus explicites présentent certaines similitudes en ce sens qu'ils subdivisent une architecture en modules relativement indépendants, qui dialoguent *via* les signaux. Ils possèdent tous deux un domaine privé, donc une zone

déclarative qui permet de créer des objets dont les noms ne sont connus qu'à l'intérieur du module.

Ils diffèrent sur un point majeur :

- Les blocs sont des architectures locales, leur intérieur continue donc à faire partie du monde concurrent. Ils fournissent un moyen de réunir des instructions concurrentes afin de leur faire partager des déclarations locales, invisibles du reste de la description. Ils peuvent bien entendu être imbriqués les uns dans les autres, de façon à créer une hiérarchie interne à l'architecture<sup>1</sup>.
- Les processus sont des modules de programme qui constituent des algorithmes séquentiels, au sens informatique du terme ; ils possèdent un jeu d'instructions propre au monde séquentiel. Leurs données privées sont des variables, jamais des signaux. Ils peuvent décrire des opérateurs tant séquentiels que combinatoires. L'instruction `process` étant, en elle-même, une instruction concurrente, les processus ne peuvent pas être imbriqués.

## ➤ Blocs

La syntaxe de création d'un bloc reprend celle d'un composant, donc celle d'une entité, et celle d'une architecture locale. Elle rajoute la possibilité de rendre conditionnelle l'exécution de certaines instructions par une expression booléenne de garde :

```

block_statement ::=
    block_label : block [ ( guard_expression ) ] [ is ]2
        [ generic ( generic_list ) ;
        [ generic map ( association_list ) ; ] ]
        [ port ( port_list ) ;
        [ port map ( association_list ) ; ] ]
        block_declarative_part
    begin
        block_statement_part ]
    end block [ block_label ] ;

```

Si une expression de garde est présente, elle crée un signal local, de type booléen, nommé `GUARD`, qui prend la valeur de l'expression. Ce signal ne doit naturellement pas être l'objet d'une affectation dans le corps du bloc. Les affectations gardées intérieures à ce bloc ne seront exécutées que si la condition de garde est vraie. Si la condition de garde est fausse, la valeur prise par le signal dépend de sa catégorie : maintien dans l'état pour un signal ordinaire, déconnexion pour un signal gardé.

1. La différence principale entre un composant instancié et un bloc est que le premier est externe, il fait référence à une unité de conception indépendante, qui peut être écrite dans un fichier à part, compilée séparément et être intégrée dans une librairie. Le bloc ne peut pas être sorti de l'architecture dont il dépend.
2. Le mot `is` est rajouté par la norme 1993, de façon optionnelle, pour rendre l'instruction homogène avec la spécification d'entité.

*Exemple :*

L'exemple ci dessous modélise un multiplexeur à sorties trois états au moyen d'un bloc :

```

library ieee ;
use ieee.std_logic_1164.all ;

entity muxN_1_tri is
  generic ( dimension : integer := 4 ) ;
  port( entree : in std_logic_vector(dimension -1 downto 0) ;
        oe : in bit ;
        sel : in integer range 0 to dimension - 1 ;
        sort : out std_logic ) ;
end muxN_1_tri ;

architecture essai_bloc of muxN_1_tri is
  signal interne : std_logic bus ;
begin
  sort <= interne ;
  mux : block (oe = '1') is -- (oe = '1') est booléenne
  begin
    interne <= guarded entree(sel) ;
  end block mux ;
end essai_bloc ;

```

Dans cet exemple, la catégorie bus du signal interne assure que, quand l'instruction d'affectation gardée du bloc est inhibée, le signal reprend sa valeur de déconnexion qui est l'état haute impédance ('Z'). Cette valeur est définie dans la fonction de résolution du sous-type `std_logic`<sup>1</sup>. Si cette catégorie était `register`, le signal de sortie serait maintenu à sa valeur précédente, par un mécanisme de type *latch*, quand l'expression de garde vaut `FALSE`.

Le programme ci-dessous réalise la même fonction, avec un bloc qui est décrit comme un sous-ensemble autonome, avec ses ports d'accès et ses propres paramètres génériques :

```

architecture essai_bloc1 of muxN_1_tri is
begin
  mux : block (oe = '1') is
    generic( taille : integer := 2 ) ;
    generic map( taille => dimension ) ;
    port ( adresse : in integer range 0 to taille - 1 ;
          din : in std_logic_vector(taille -1 downto 0) ;
          sort : out std_logic ) ;
    port map (adresse => sel, din =>entree , sort => sort ) ;
    signal interne : std_logic bus := 'Z' ;
  end block mux ;
end essai_bloc1 ;

```

1. Les fonctions de résolution servent également, rappelons-le, à trancher les conflits générés par des signaux pilotés par plusieurs sources ; nous aurons l'occasion d'en reparler.



```

begin
    interne <= guarded din(adresse) ;
    sort <= interne ;
end block mux ;
end essai_bloc1 ;

```

Notons qu'à l'heure actuelle les outils de synthèse n'utilisent pas toujours la totalité de l'instruction `block`, notamment en ce qui concerne les signaux gardés. Avec certains outils de synthèse le découpage en blocs peut être conservé lors de la traduction du programme en schéma. Cela permet d'affiner le pilotage de l'optimiseur en fixant des politiques spécifiques pour certains blocs. C'est un avantage non négligeable.

### » Processus

Les processus jouent, le lecteur attentif s'en sera sans doute rendu compte, un rôle central dans la sémantique du langage. Un processus est une instruction structurée concurrente dont le contenu est une suite d'instructions séquentielles. Nous décrivons ici l'aspect instruction concurrente, le contenu sera développé au paragraphe suivant.

Rappelons au préalable que chaque instruction d'affectation concurrente de signal crée un processus implicite. La structure interne de ce processus n'est pas visible, on sait simplement qu'il est activé dès qu'un événement affecte l'un quelconque des signaux qui apparaît dans la partie droite de l'instruction d'affectation. La création d'un processus explicite permet de dissocier les signaux qui sont les opérandes des expressions des signaux qui rendent le processus actif.

La syntaxe de l'instruction est voisine de celle de la création de bloc, avec un contenu sémantique différent :

```

process_statement ::=
    process_label :
        [ postponed ]1 process [ ( sensitivity_list ) ] [ is ]
            process_declarative_part
        begin
            process_statement_part ]
        end [ postponed ] process [ process_label ] ;

```

La liste de sensibilité contient la liste des signaux dont un événement doit réveiller un processus en sommeil<sup>2</sup>. Elle n'est pas obligatoire, mais si elle est absente le processus doit contenir une instruction explicite d'attente, `wait`. Liste de sensibilité et instruction `wait` sont exclusives l'une de l'autre, elles ne doivent en aucun cas être présentes simultanément. Certains outils de synthèse ont une lecture très approximative de la liste de sensibilité, ce point peut réserver des surprises désagréables.

1. Le mot `postponed` (ajourné), optionnel, est rajouté par la norme 93. Il indique au simulateur que le `process`, s'il doit être activé, ne doit l'être qu'au dernier cycle delta du temps courant. Par défaut un processus n'est pas ajourné.
2. Soulignons qu'un processus connaît tous les signaux de l'architecture englobante. La liste de sensibilité ne joue en aucun cas un rôle similaire à une liste d'arguments passés à une procédure. Cette confusion est fréquente chez les débutants.

La zone de déclaration contient la description du domaine privé du processus, rappelons que peuvent y figurer des variables, mais pas des signaux.

La zone d'instructions contient des instructions exécutées séquentiellement, au sens informatique du terme. Cela ne présuppose pas que le processus décrive un opérateur séquentiel. Nous avons précédemment décrit un multiplexeur, un opérateur purement combinatoire, au moyen d'un processus.

### Exemples :

La suite de ce texte est consacrée à l'étude des processus, nous nous contenterons ici de donner deux exemples très simples, mais qui méritent réflexion.

Traduisons un petit programme déjà rencontré par un processus :

```
architecture sequence of op_abarb is
begin
  action : process (a, b)
    variable abar : bit ;
  begin
    abar := not a ;
    abarb <= abar and b ;
  end process action ;
end sequence ;
```

Notons en passant la différence entre les symboles d'affectation de variables ( := ) et de signaux ( <= ).

Les signaux observables à l'issue d'une simulation ont exactement la même allure, en fonction du temps, que précédemment, ce qui est rassurant. L'examen attentif de la liste des événements montre que l'élaboration interne de ce résultat n'est pas la même :

ns	delta	a	b	abarb
0	+0	0	0	0
0	+1	1	0	0
10	+0	0	1	0
10	+1	0	1	1
20	+0	1	0	1
20	+1	1	0	0

Les événements internes ont disparu. En fait, la première version de ce programme (page 55) créait deux processus interagissant, ici il n'en reste qu'un.

La même transformation appliquée à la boucle infernale conduit, bien sûr, à une boucle infernale, mais qui a besoin de moins de cycles delta pour osciller :

ns	delta	b	abarb
0	+0	0	0
20	+0	1	0
20	+1	1	1
20	+2	1	0
20	+3	1	1
20	+4	1	0

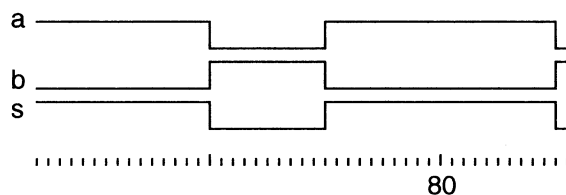
etc.

Notre deuxième exemple, issu d'un livre récent, semble *a priori* très simple, mais l'est moins que son auteur ne le laisse penser :

```
entity exemple_proc is
  port ( a : in bit ; b : out bit );
end exemple_proc ;

architecture simple of exemple_proc is
  signal s : bit ;
begin
  process (a)
  begin
    s <= a ;
    b <= s ;
  end process ;
end simple ;
```

Un événement sur *a* rend le processus actif. Le processus demande alors au noyau de copier la nouvelle valeur de *a* dans *s* et, dans le même temps, de transférer la valeur courante de *s* en sortie. Cette valeur est celle que le processus trouve à son réveil, c'est-à-dire l'ancienne valeur de *a*, qui ne peut être que son complément, pour un signal binaire. Ce que confirme bien une simulation (figure 2-9), où apparaît un comportement de « registre à décalage », actif sur les deux fronts de *a*, qui transfère en sortie sa propre horloge. Quand *a* est binaire le résultat est le complément de *a*.



**Figure 2-9** Un registre à décalage curieux.

Cet exemple illustre bien le mécanisme de la chronologie induite par les événements. Par contre il laisse sans voix les outils de synthèse : l'un fournit un résultat faux (*b* et *a* identiques), un autre, plus prudent, refuse de synthétiser cet objet pathologique, actif sur deux fronts et qui recopie sa propre horloge. Aucun ne détecte le comportement purement combinatoire du montage. Si *a* n'est pas binaire le comportement n'est évidemment plus combinatoire, mais encore moins synthétisable.

### 2.3.2 Le corps d'un processus : le monde séquentiel

La syntaxe des instructions qui interviennent dans un processus ne pose guère de problème pour qui a pratiqué un tant soit peu un langage de programmation classique. VHDL est un langage structuré en blocs, comme C ou PASCAL. L'investissement à faire est au niveau sémantique, le sens de ce que l'on écrit n'a souvent qu'un lointain rapport avec celui d'un langage procédural.

### a) Une boucle sans fin contrôlée par des événements

Vu par un informaticien, un processus est une boucle sans fin. Au lancement du simulateur tous les processus sont activés, charge au programmeur de prévoir un mécanisme de mise en sommeil jusqu'à ce que survienne un événement. Cette mise en sommeil peut se faire explicitement, par une instruction `wait`, qui indique au processus qu'il doit attendre que quelque chose se produise, ou par une liste de sensibilité qui indique les signaux dont le changement de valeur provoque le réveil du processus. Quelques remarques importantes doivent être présentes à l'esprit :

- Un processus qui ne possède ni liste de sensibilité ni `wait` monopolise complètement le simulateur, sans que rien ne se passe, le temps n'évolue pas. Un tel piège est, quand il est détectable, signalé à la compilation, mais n'est pas une erreur.
- Le noyau attribue à tout signal qui est l'objet d'une affectation dans un processus un pilote (*driver*) unique, même si plusieurs instructions d'affectation figurent dans le code du processus. Les affectations séquentielles peuvent donc être multiples, même pour les signaux ordinaires. Si le même signal est l'objet d'affectations dans différents processus, il se voit affectés autant de pilotes, ce signal doit donc être résolu car il y a conflit.
- Les affectations séquentielles de signaux ne prennent effet que quand le processus se remet en attente. En fait elles correspondent à une transaction gérée par le noyau. Pendant l'exécution du processus aucun signal ne change de valeur, ce pendant dure de toute façon un temps (virtuel) nul.
- Entre deux activations d'un processus qui affecte une valeur à un signal, cette valeur est mémorisée. Tout non-dit est interprété comme une demande de non-modification, c'est-à-dire une mémoire.

#### ➤ L'instruction `wait`

L'instruction `wait` suspend l'exécution d'un processus jusqu'à ce qu'un événement, une condition ou une clause de temps écoulé (*time out*) soit vraie. Si aucune clause de réveil n'est stipulée, le processus s'arrête définitivement.

```

=====
[ label : ]
wait [ on sensitivity_list ] [ until condition ] [ for time_expression ] ;
=====

```

La liste de sensibilité spécifie les noms des signaux à surveiller. Si un événement survient sur l'un de ces signaux, la condition est examinée. Si cette condition est vraie le processus est activé, si non il reste suspendu. Il n'y a aucun lien obligatoire entre les signaux de la liste de sensibilité et ceux qui interviennent dans les autres instructions du processus. Si une condition est fournie, mais pas de liste de sensibilité, cette dernière comporte tous les signaux qui interviennent dans la condition. Ainsi :

```

=====
wait until hor = '1' ;
=====

```

est équivalent à :

```

=====
wait on hor until hor = '1' ;
=====

```

Le processus correspondant est activé par les fronts montants du signal `hor`, modélisant un opérateur synchrone (*rising edge triggered*).

La dernière clause indique le temps maximum qui sépare deux activités du processus, cela n'a évidemment de sens qu'en simulation.

En simulation plusieurs instructions `wait` peuvent apparaître dans le corps d'un processus. Les outils de synthèse n'en acceptent généralement qu'une seule, placée au début ou à l'extrême fin du processus. Cette instruction doit, de surcroît, correspondre à un opérateur physiquement réalisable dans le circuit cible ; il est impossible, par exemple, de synthétiser une fonction qui réagit à la fois sur les fronts montants et descendants d'un signal.

### ➤ La liste de sensibilité

Une liste de sensibilité est équivalente à un `wait on ...` placé en fin de processus. Ainsi :

```
process (a)
begin
    s <= a ;
    b <= s ;
end process ;
```

est équivalent à :

```
process
begin
    s <= a ;
    b <= s ;
    wait on a ;
end process ;
```

Ce placement, en fin de processus, de l'instruction `wait` équivalente à la liste de sensibilité peut provoquer une différence marginale de fonctionnement entre le début d'une simulation et la mise en route réelle d'un circuit synchrone : dans le circuit réel la situation n'évolue pas tant que le signal d'horloge ne présente pas de front actif, alors que le processus équivalent s'exécute une fois à l'initialisation du simulateur.

### b) Signaux et variables

Signaux et variables ont un comportement radicalement différent dans un processus.

#### ➤ Réaction immédiate : les variables

L'affectation ( symbole `:=` ) d'une valeur à une variable :

```
process
variable a : integer := expression_constant ;
...
begin
    a := expression ;
    ...
```

prend effet immédiatement, comme ce serait le cas dans un langage de programmation ordinaire.

Son initialisation, par une expression précisée dans la zone de déclaration, est faite une seule fois, au temps zéro, au lancement du simulateur. Cette initialisation n'a donc aucun effet en synthèse.

Une variable conserve sa valeur entre deux activités du processus. Ce point peut conduire à des difficultés dans du code synthétisable, puisqu'une variable n'a aucune existence physique dans le circuit cible. En synthèse, un objet qui conserve sa valeur, entre deux activation d'un processus, est une cellule mémoire du circuit. Il est donc préférable de lui attacher un signal plutôt qu'une variable. Les outils de synthèse réagissent différemment face à des variables qui ne peuvent pas disparaître lors de l'élaboration du circuit.

Pour supprimer tout danger de mémorisation, une méthode simple consiste à affecter systématiquement une valeur par défaut, dans la zone des instructions du processus, à toutes les variables utilisées.

#### ► Réaction différée : les signaux

L'instruction séquentielle d'affectation d'une valeur à un signal ne peut prendre que sa forme simple, les tests éventuels étant des instructions à part entière :

```
[label :] target <= [delay_option] waveform ;
delay_option ::= transport | [reject time_express] inertial
```

Rappelons que l'affectation d'une valeur à un signal ne prend effet que quand le processus se remet en attente, elle définit une forme d'onde prévue dans le futur (*projected waveform*). Ce point assure l'indivisibilité des actions sur les signaux. Plusieurs instructions d'affectation du même signal sont légales, seule la dernière exécutée sera effectivement prise en compte.

La cible peut être le nom d'un signal ou, mais cette construction est plutôt rare, un agrégat.

#### ► Temps et chronologie des événements

La différence entre variables et signaux apparaît clairement quand on modélise des opérateurs séquentiels synchrones. Les deux architectures qui suivent ont des écritures proches, mais des comportements temporels très différents :

```
entity registre is
port( hor, vin : in bit ;
      vout0, vout1 : out bit );
end registre ;

architecture decalage of registre is
  signal v0, v1 : bit ;
begin
  vout0 <= v0 ;
  vout1 <= v1 ;
process
  variable temp : bit ;
```

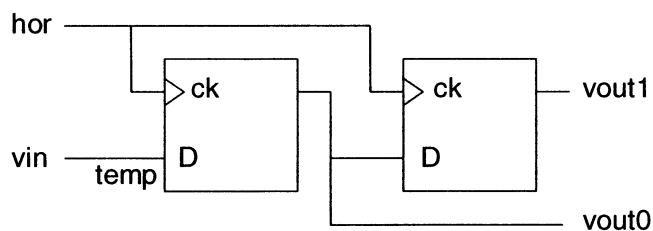
```

begin
    wait until hor = '1' ;
    temp := vin ;
    v0 <= temp ;
    v1 <= v0 ;
end process ;
end decalage ;

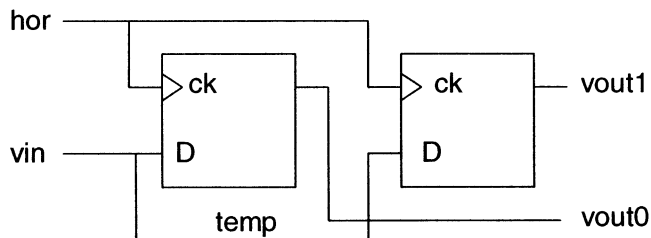
architecture parallele of registre is
    signal v0, v1 : bit ;
begin
    vout0 <= v0 ;
    vout1 <= v1 ;
    process
        variable temp : bit ;
    begin
        wait until hor = '1' ;
        temp := vin ;
        v0 <= temp ;
        v1 <= temp ;
    end process ;
end parallele ;

```

La première décrit un registre à décalage, vout1 est égal à vout0 retardé d'une période d'horloge. La seconde décrit un registre de deux bascules D qui se chargent en parallèle à la même valeur, vin, à chaque front d'horloge, conformément au schéma de la figure 2-10.



**architecture décalage**



**architecture parallèle**

**Figure 2-10** Variables et signaux.

Les deux architectures sont synthétisables, mais ne réalisent pas du tout la même fonction.

### c) Instructions séquentielles

Comme tout langage de programmation, VHDL possède des instructions séquentielles de contrôle, tests et boucles dont la syntaxe est sans surprise. Elles sont au coeur des descriptions comportementales des circuits, plus que jamais, dans ce type de description, il est essentiel de toujours avoir présente à l'esprit l'architecture du circuit que l'on décrit, ses signaux d'horloge, ses registres, ses blocs combinatoires.

#### ► Tests (*if, case*)

Les deux catégories de tests utilisables en VHDL sont les tests logiques et les sélections de valeurs portant sur un type discret.

#### *Tests logiques*

Ils permettent d'exécuter des blocs d'instructions en fonction des résultats de l'évaluation de conditions booléennes. Un traitement par défaut peut être spécifié pour le cas où toutes les conditions sont fausses (*else*) :

```
[label : ] if condition then
    instructions
{ elsif condition then
    instructions }
[ else
    instructions ]
end if [label] ;
```

Les conditions sont évaluées dans l'ordre d'écriture, des différents blocs d'instructions, un seul, au maximum, est exécuté, en fonction des tests. L'ordre d'écriture permet donc de créer une priorité entre des commandes. Par exemple :

```
if raz = '0' then
    compte <= 0 ;
elsif en = '1' then
    compte <= (compte + 1) mod MAX ;
end if ;
```

décrit un compteur (de 0 à MAX - 1) avec un signal de remise à zéro (*raz*) prioritaire sur le signal d'autorisation de comptage (*en*). Quand les deux signaux de commande sont inactifs, aucune action n'est spécifiée, le signal *compte* conserve sa valeur précédente.

#### *Sélections*

Une sélection spécifie des blocs d'instructions à effectuer en fonction de la valeur d'une expression testée. L'expression doit être d'un type discret, ou un vecteur d'éléments de type caractère dont la dimension est fixe. Toutes les valeurs possibles de l'expression testée doivent apparaître une fois et une seule dans la suite des alternatives.



Le mot clé `others` permet de regrouper toutes les valeurs non précisées explicitement dans la dernière alternative :

```
[label :] case expression is
    when choices => instructions
    { when choices => instructions }
end case [label] ;
choices ::= choice { | choice }
choice ::= static_expression | discrete_range | others
```

Le compteur précédent peut être décrit par une sélection, bien que la lisibilité du code en souffre :

```
case (raz & en) is -- & : concaténation
    when "00" | "01" => compte <= 0 ;
    when "11"        => compte <= (compte + 1) mod MAX ;
    when others      => null ;
end case ;
```

L'instruction `null`, utilisée ici pour respecter la contrainte d'exhaustivité des choix, ne provoque aucune action, le signal `compte` ne change pas de valeur.

### » Mémoire implicite

Nous avons vu que pour chaque signal, sujet à une affectation dans un processus, le compilateur crée un pilote. Il peut arriver, les deux écritures du compteur précédent le montrent, qu'à l'issue de l'activation d'un processus, aucune action ne soit spécifiée pour un signal piloté par ce processus. Dans ce cas le signal ne change pas de valeur, il est donc mémorisé. Quand on décrit un opérateur séquentiel synchrone, cette mémoire par défaut est un avantage, elle permet de ne spécifier que les transitions entre états, par exemple. Quand on décrit un opérateur asynchrone ou, pire, combinatoire, cette mémoire par défaut peut être un piège. Toute omission dans les tests se traduit par la création d'une mémoire, ce que l'auteur du programme n'a pas toujours souhaité.

### » Boucles

La première boucle, la boucle temporelle fondamentale, est le processus lui-même. L'oubli de ce mécanisme de base de la sémantique d'un programme VHDL est la source de bien des confusions.

Si, donc, on souhaite créer une évolution temporelle décrite par une boucle, il n'y a rien à rajouter, chaque processus d'un programme est une boucle sans fin dans laquelle il ne reste qu'à préciser des conditions d'attente d'événements.

Par exemple, pour créer un compteur modulo 10, il n'est nullement besoin de faire appel à une instruction spécifique de boucle, il suffit de créer un processus :

```
architecture comporte of decade is
    signal compte : integer range 0 to 9 ;
begin
    compteur : process
```

```

begin
    wait until horloge = '1' ;
    compte <= (compte + 1) mod 10 ;
end process compteur ;
end comporte ;

```

Alors quand donc a-t-on besoin des boucles ? Le plus souvent pour créer des boucles spatiales et non temporelles.

Nous entendons par boucles spatiales une série d'opérations, qui peuvent bien se représenter par un algorithme, en raison de leur répétitivité, et qui se déroulent au même instant, à différents endroits physiques d'un schéma. Prenons comme exemple un générateur de parité, un circuit qui étant donné un vecteur d'entrée composé d'éléments binaires, fournit en sortie un signal binaire qui vaut '1' si le nombre de '1' du signal d'entrée est impair, par exemple, et '0' autrement :

```

ENTITY ouex IS
    generic(taille : integer := 8) ;
    PORT ( a : IN BIT_VECTOR(0 TO taille - 1) ;
          s : OUT BIT );
END ouex;

ARCHITECTURE parite of ouex is
BEGIN
    process(a)
        variable parite : bit ;
    begin
        parite := '0' ;
        FOR i in a'range LOOP
            if a(i) = '1' then
                parite := not parite;
            end if;
        END LOOP;
        s <= parite;
    end process;
END parite;

```

La boucle utilisée ici, une boucle *for*, sert à explorer tous les éléments du vecteur d'entrée. Le circuit correspondant est purement combinatoire<sup>1</sup>, la boucle sert à itérer une opération sur plusieurs éléments du circuit. La boucle est effectuée dans sa totalité à chaque activation du processus, en un temps (virtuel) nul.

1. La version proposée dans notre précédent ouvrage pour ce même opérateur contenait, *errare humanum est*, une utilisation non portable, car erronée, de la variable *parite* qui n'était pas initialisée à chaque activation du processus. L'aspect savoureux de la chose est que l'erreur n'était pas décelée par le compilateur utilisé.

L'exemple qui précède utilise une boucle `for`, qui n'est que l'une des boucles connues du langage. La syntaxe générale de construction d'une boucle est :

```
[label] : [iteration_scheme] loop
    instructions
end loop [label] ;
iteration_scheme ::=
    while condition | for parameter_specification
parameter_specification ::= identifieur in discrete_range
```

Si le schéma d'itération est absent, la boucle est éternelle, ce qui suppose évidemment un autre mécanisme de sortie, par l'instruction `exit`, par exemple.

Le paramètre d'une boucle `for` est déclaré implicitement par sa spécification, sa valeur ne peut pas être modifiée dans les instructions du corps de la boucle, et il est inconnu en dehors de la boucle.

Deux instructions<sup>1</sup> permettent de modifier le déroulement naturel d'une boucle, les instructions `next` et `exit` :

```
[label] : next [loop_label] [when condition] ;
[label] : exit [loop_label] [when condition] ;
```

La première provoque une nouvelle itération, si cela doit être, sautant ainsi les instructions suivantes du corps de la boucle ; la seconde provoque une sortie de la boucle. Ces deux instructions doivent être à l'intérieur de la boucle contrôlée ; en cas de boucles imbriquées, si l'étiquette n'est pas spécifiée, elles contrôlent la boucle la plus interne.

Nous avons souligné que pour beaucoup d'outils de synthèse l'instruction `wait` d'un processus, si elle existe, est unique et doit être la première ou la dernière instruction du corps d'un processus. Il en résulte que pour ces outils il est impossible de bloquer le déroulement d'une boucle. Autrement dit, les boucles synthétisables ne sont jamais temporelles, elles ont un temps d'exécution toujours nul et décrivent une structure purement spatiale.

Terminons cette première description des boucles par l'utilisation d'une boucle `for` pour modéliser un registre à décalage simple, à entrée série et sorties parallèles :

```
ENTITY sipo IS
    generic(taille : integer := 8) ;
    PORT (hor, serin : in bit ;
          etat : buffer BIT_VECTOR(0 TO taille - 1) );
END sipo;

ARCHITECTURE boucle of sipo is
BEGIN
    process
```

1. Outre le retour d'un sous-programme, que nous aborderons ultérieurement.

```

begin
  wait until hor = '1' ;
  FOR i in etat'range LOOP
    if i = 0 then
      etat(i) <= serin ;
    else
      etat(i) <= etat(i - 1) ;
    end if ;
  END LOOP ;
end process ;
END boucle ;

```

Ce programme illustre l'intérêt majeur des boucles. Le même code peut décrire un registre de 8 bascules, c'est le cas par défaut, ou un registre de 64 bascules, ou de toute autre taille, en modifiant à l'instanciation du composant correspondant la valeur du paramètre générique *taille*.

### 2.3.3 Modélisation des opérateurs logiques combinatoires

VHDL offre un nombre important de possibilités pour décrire un opérateur logique. Pour illustrer ce point nous prendrons un circuit classique des catalogues TTL : le décodeur 74138. Il s'agit d'un décodeur  $3 \rightarrow 8$ , dont les sorties (O0 à O7) sont actives au niveau bas, qui comporte trois entrées binaires d'adresse (C, B et A) et trois entrées de validation (G1 active au niveau haut, G2A et G2B actives au niveau bas).

Quelle que soit l'architecture, la déclaration d'entité est la même :

```

entity dec_74_138 is
  port ( A, B, C, G1, G2A, G2B : in bit ;
         O : out bit_vector (0 to 7) );
end dec_74_138 ;

```

Nous présentons ci-dessous quelques solutions d'architectures, les premières font appel à des instructions concurrentes, les secondes à des processus explicites, donc des algorithmes séquentiels bien que l'opérateur modélisé soit purement combinatoire.

#### a) Par des instructions concurrentes

Outre les descriptions structurelles, que nous laissons au lecteur le soin d'élaborer, les instructions concurrentes peuvent utiliser des tables de vérité, des équations logiques traditionnelles ou des affectations structurées, dans une description flot de données.

#### ➤ Tables de vérité

Une table de vérité n'est autre qu'un tableau de constantes dont les index sont les entrées de la table et les éléments les valeurs de la table :

```

architecture table of dec_74_138 is
type table_dec is array(bit,bit,bit)
                        of bit_vector(0 to 7) ;
constant decode : table_dec :=
-- sorties : 01234567      adresses
  (((("01111111", --000
      "10111111"), --001
      ("11011111", --010
      "11101111")), --011
    (("11110111", --100
      "11111011"), --101
      ("11111101", --110
      "11111110"))); --111
signal valid : boolean ;
begin
  valid <= (G1 and not G2A and not G2B) = '1' ;
  O <= decode(C,B,A) when valid else (others => '1') ;
end table ;

```

L'aspect un peu rébarbatif de l'agrégat qui définit la table est compensé par la simplicité du corps de l'architecture. Pour comprendre la construction de cet agrégat il faut interpréter un tableau de dimension  $n$  comme un tableau de tableaux de dimension  $n - 1$ , etc. Les parenthèses définissent l'agrégat comme une composition de sous-agrégats.

### ➤ Équations logiques

Les équations logiques sont la simple traduction du logigramme du circuit décrit. Dans les cas simples il ne faut pas négliger cette possibilité, l'exemple ci-dessous est sans doute un peu à la limite de ce qui reste lisible :

```

architecture equations of dec_74_138 is
  signal valid : bit ;
begin
  valid <= G1 and not G2A and not G2B ;
  O(0) <= not (valid and not A and not B and not C) ;
  O(1) <= not (valid and A and not B and not C) ;
  O(2) <= not (valid and not A and B and not C) ;
  O(3) <= not (valid and A and B and not C) ;
  O(4) <= not (valid and not A and not B and C) ;
  O(5) <= not (valid and A and not B and C) ;
  O(6) <= not (valid and not A and B and C) ;
  O(7) <= not (valid and A and B and C) ;
end equations ;

```

Rappelons à cette occasion que les différents opérateurs logiques, agissant sur deux opérandes, ont tous la même priorité. Si des expressions contiennent des opérateurs différents, il convient de les parenthéser correctement.

### ► Flot de données

La description flot de données qui suit, bien qu'un peu longue, est très simple à construire à partir de la table de vérité de l'opérateur :

```
architecture flot of dec_74_138 is
  signal valid : bit ;
  subtype val_ad is bit_vector(0 to 3) ;
begin
  valid <= G1 and not G2A and not G2B ;
  with val_ad'(valid & a & b & c) select -- précision de type
    0 <= "01111111" when "1000",
        "10111111" when "1100",
        "11011111" when "1010",
        "11101111" when "1110",
        "11110111" when "1001",
        "11111011" when "1101",
        "11111101" when "1011",
        "11111110" when "1111",
        "11111111" when others ;
end flot ;
```

On notera que la concaténation d'objets de type bit, permet de construire un vecteur anonyme d'éléments de ce type.

### b) Par un algorithme séquentiel

Dans la description d'un opérateur combinatoire, un algorithme séquentiel présente parfois l'avantage de souligner, par l'ordre d'écriture, les échelles de priorités des commandes de l'opérateur. Dans le cas qui nous intéresse, la fonction décodeur n'est à examiner que si l'opérateur est validé :

```
architecture comporte of dec_74_138 is
begin
  decode : process (A, B, C, G1, G2A, G2B)
    variable valid : boolean ;
    subtype adresse is bit_vector(0 to 2) ;
  begin
    valid := (G1 and not G2A and not G2B) = '1' ;
    if not valid then
      0 <= (others => '1') ;
    else
      case adresse'(A & B & C) is
        when "000" => 0 <= "01111111" ;
        when "100" => 0 <= "10111111" ;
        when "010" => 0 <= "11011111" ;
        when "110" => 0 <= "11101111" ;
        when "001" => 0 <= "11110111" ;
        when "101" => 0 <= "11111011" ;
        when "011" => 0 <= "11111101" ;
        when "111" => 0 <= "11111110" ;
      end case ;
    end if ;
  end process decode ;end comporte ;
```

L'expression testée par l'instruction `case` doit être qualifiée (son sous-type précisé) car son type doit être défini localement, indépendamment du contexte dans lequel l'instruction est utilisée. Le caractère combinatoire de l'opérateur est assuré par les deux branches de l'alternative traitées par l'instruction `if`. Quel que soit le résultat de l'expression testée une valeur est affectée aux sorties.

Notre dernière version du décodeur relève un peu de l'utilisation d'un marteau pilon pour écraser une mouche, mais elle illustre, par anticipation, l'existence de bibliothèques de conversion entre les vecteurs d'éléments binaires et les nombres :

```
library ieee ;
use ieee.numeric_bit.all ;

architecture calcule of dec_74_138 is
begin
  decode : process (A, B, C, G1, G2A, G2B)
    variable valid : boolean ;
    variable index : integer range 0 to 7 ;
  begin
    valid := (G1 and not G2A and not G2B) = '1' ;
    -- appel de la fonction to_integer() de la bibliothèque :
    index := to_integer(unsigned'(C & B & A)) ;
    0 <= (others => '1') ;
    if valid then
      0(index) <= '0' ;
    end if ;
  end process decode ;
end calcule ;
```

Le principe consiste à traiter les trois entrées d'adresse comme un nombre qui indique le numéro de la sortie qui doit être active, ce qui est somme toute assez naturel. Une solution équivalente aurait été de déclarer dans l'entité l'entrée d'adresse comme un nombre. Mais nous aurions perdu la correspondance des noms avec la notice du composant.

On notera, dans ce dernier programme, la valeur par défaut adoptée pour toutes les sorties. Cette précaution garantit le caractère combinatoire de l'opérateur bien que l'instruction `if` ne traite pas explicitement tous les résultats du test.

### 2.3.4 Modélisation des opérateurs séquentiels

La différence entre un opérateur combinatoire et un opérateur séquentiel est que ce dernier est doué de mémoire. VHDL est un langage de description qui privilégie cette notion de mémoire, ce qui le rend particulièrement efficace pour modéliser des opérateurs séquentiels complexes. La contrepartie de cette efficacité est que l'utilisateur novice risque de créer plus de cellules mémoires que nécessaire, et que les commandes de ces cellules risquent d'être quelque peu anarchiques. Nous commencerons par explorer, sur un exemple extrêmement simple, le passage d'un opérateur combinatoire à un opérateur séquentiel, asynchrone puis synchrone.

Nous n'envisagerons ici que des modélisations comportementales qui utilisent des processus explicites. Il est bien sûr possible de créer des fonctions séquentielles à partir de composants, dans une description structurelle, mais cela ne fait que repousser le problème. Il est également possible de créer des opérateurs séquentiels au moyen d'instructions concurrentes, affectations conditionnelles et blocs gardés, mais cette méthode, moins naturelle, n'est pas toujours acceptée par les outils de synthèse. De toute façon elle ne se prête qu'à des opérations tellement simples qu'il n'y a pas de confusion possible.

#### a) Opérateurs synchrones et asynchrones

La différence entre un opérateur asynchrone et un opérateur synchrone réside dans la commande de mémorisation. Le premier conserve une trace de fonctionnement combinatoire, plusieurs signaux sont susceptibles de le faire changer d'état, en fonction de leurs niveaux. Le second ne change d'état que suite à un événement précis, front montant ou descendant, qui implique un signal unique : l'horloge. Toutes les autres entrées d'un opérateur synchrone ne servent qu'à déterminer la nature de l'opération qui sera effectuée lors de la transition active de l'horloge.

L'immense avantage des opérateurs synchrones est qu'ils permettent de fixer des règles simples qui assurent un fonctionnement sans aléa<sup>1</sup>. Schématiquement, ces règles peuvent se résumer en une phrase : « on ne change pas les commandes en même temps que l'on en demande l'exécution ». En clair, les commandes doivent être stables un peu avant, et éventuellement un peu après, le front actif de l'horloge.

#### ► Signaux et cellules mémoires

Prenons un exemple simple, une porte ET, opérateur combinatoire s'il en est :

```
entity comb_seq is
  port ( e1, e2 : in bit ;
         sort : out bit ) ;
end comb_seq ;

architecture comporte of comb_seq is
begin
  et : process (e1,e2)
  begin
    if( e1 = '1' ) then
      sort <= e2 ;
    else
      sort <= '0' ;
    end if ;
  end process et ;
end comporte ;
```

1. Les choses ont un coût : la consommation. Un opérateur synchrone a un fonctionnement dynamique, même s'il ne fait rien, à chaque front d'horloge il interprète ses commandes, quelles qu'elles soient. Cette activité permanente se paye par une consommation plus importante qu'un circuit réellement au repos.



Nous laissons au lecteur le soin de contrôler la justesse du code. L'aspect combinatoire de l'opérateur se vérifie aisément : quels que soient les événements portant sur e1 ou e2, la sortie a une valeur explicite.

Modifions très légèrement le code du processus :

```
architecture comporte of comb_seq is
begin
  d_latch : process (e1,e2)
  begin
    if( e1 = '1' ) then
      sort <= e2 ;
    end if ;
  end process d_latch ;
end comporte ;
```

Seule a disparu la deuxième alternative du test. Quand e1 = '0', aucune valeur n'est précisée pour la sortie. Elle conserve donc sa valeur précédente, l'opérateur est une bascule D *latch*. Le caractère asynchrone de l'opérateur provient de la liste de sensibilité, e2 peut provoquer le changement de valeur de la sortie tant que e1 est au niveau '1'.

Modifions encore très légèrement le code de notre programme, supprimons la possibilité d'un changement de valeur de la sortie par un événement sur e2 :

```
architecture comporte of comb_seq is
begin
  d_edge : process (e1)
  begin
    if( e1 = '1' ) then
      sort <= e2 ;
    end if ;
  end process d_edge ;
end comporte ;
```

La liste de sensibilité du processus ne contient plus que le signal e1. À chaque événement lié à ce signal suivi d'un niveau '1', donc, comme ce signal est binaire, à chaque front montant, la sortie prend la valeur fixée par e2. Dans tous les autres cas, front descendant de e1, modifications de e2, la sortie conserve sa valeur précédente. C'est la définition même d'une bascule D *edge*.

Si la liste de sensibilité est remplacée par une instruction wait équivalente le résultat est le même. On peut même en profiter pour éliminer des tests inutiles :

```
entity comb_seq is
  port ( e1, e2 : in bit ;
         s_et, s_latch, s_edge : out bit ) ;
end comb_seq ;

architecture comporte of comb_seq is
begin
  et : process
```

```

begin
  wait on e1, e2 ;
  if( e1 = '1' ) then
    s_et <= e2 ;
  else
    s_et <= '0' ;
  end if ;
end process et ;

latch : process
begin
  wait on e1, e2 until e1 = '1' ;
  s_latch <= e2 ;
end process latch ;

edge : process
begin
  wait on e1 until e1 = '1' ;
  s_edge <= e2 ;
end process edge ;
end comporte ;

```

Les différences entre les codes sources sont subtiles, il faut le reconnaître, mais les comportements des opérateurs sont radicalement différents. Ces exemples méritent réflexion. Quand ils sont bien assimilés, le langage s'avère très efficace ; tant qu'ils n'ont pas été compris de mauvaises surprises attendent l'utilisateur.

Les outils de synthèse ont parfois, souvent, un comportement déroutant face à des constructions qui jouent entièrement sur les listes de sensibilités et les types des signaux<sup>1</sup>. Deux précautions valent mieux qu'une, une bonne habitude consiste à préciser de façon tout à fait explicite les signaux d'horloge, quitte à introduire une redondance dans les tests en rajoutant explicitement l'attribut 'event, même là où la sémantique *stricto sensu* du langage ne l'imposerait pas.

### ➤ Horloges

Pour modéliser un opérateur entièrement synchrone trois méthodes semblent acceptées par tous les outils de synthèse et de simulation :

- l'utilisation d'une instruction `wait` unique,
- l'introduction d'un test explicite d'événement,
- l'utilisation du type `std_logic` et d'une fonction `rising_edge` ou `falling_edge` de la librairie `ieee`.

Les deux programmes ci-après résument ces méthodes pour modéliser une bascule *D edge* :

1. Le dernier programme proposé, par exemple, est amusant. Il est accepté en synthèse par deux compilateurs qui ne voient, à tort, qu'une porte et deux bascules *edge*.

```

entity d_edge is
    port ( d,hor : in bit ;
           s : out bit) ;
end d_edge;

architecture d_wait of d_edge is
begin
process
begin
wait until hor = '1';
    s <= d ;
end process ;
end d_wait ;

architecture d_liste of d_edge is
begin
process (hor)
begin
    if hor = '1' and hor'event then
        s <= d ;
    end if ;
end process ;
end d_liste ;

```

Et en utilisant la librairie ieee :

```

library ieee ;
use ieee.std_logic_1164.all ;

entity d_edge is
    port ( d,hor : in std_logic ;
           s : out std_logic) ;
end d_edge;

architecture d_ieee of d_edge is
begin
process (hor)
begin
    if rising_edge(hor) then
        s <= d ;
    end if ;
end process ;
end d_ieee ;

```

Si l'on souhaite modéliser un opérateur séquentiel normalement synchrone qui comporte des commandes asynchrones, comme une remise à zéro par exemple, la méthode classique consiste à utiliser une liste de sensibilité. L'ordre des tests établira la priorité des commandes. Une simple bascule *D edge* à remise à zéro asynchrone s'écrira, par exemple :

```

entity d_edge_raz is
    port ( d, hor, raz : in bit ;
           q : out bit) ;

```

```

end d_edge_raz ;

architecture d_liste of d_edge_raz is
begin
  process (hor,raz)
  begin
    if raz = '1' then
      q <= '0' ;
    elsif hor = '1' and hor'event then
      q <= d ;
    end if ;
  end process ;
end d_liste ;

```

Il est clair que l'on n'a pratiquement jamais besoin de modéliser des opérateurs aussi simples qu'une bascule D ; les fonctions séquentielles complexes obéissent aux mêmes principes<sup>1</sup>.

### b) Machines d'états synchrones

Les machines à nombre fini d'états jouent un rôle important dans la synthèse des fonctions logiques séquentielles. Il s'agit plus d'une manière de voir que d'une architecture spécifique, la plupart des fonctions synchrones peuvent être décrites de cette façon. Le modèle général d'une machine d'états (*finite state machine*, *FSM* en abrégé) est celui de la figure 2-11.

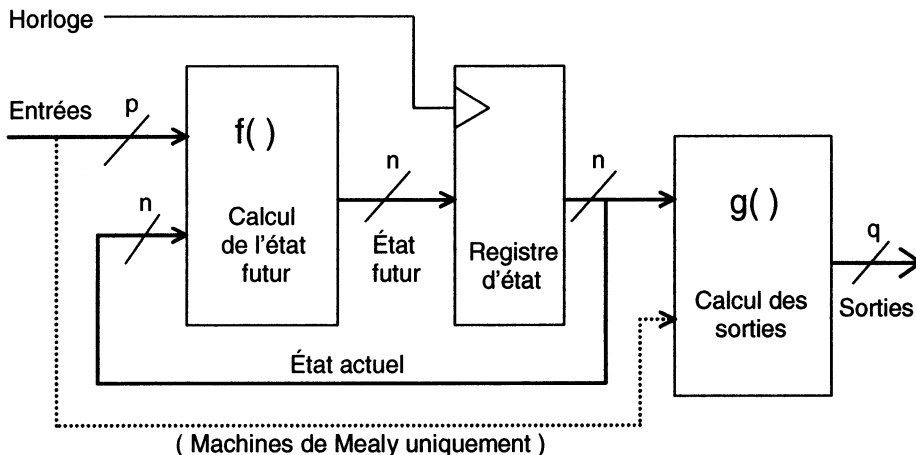


Figure 2-11 Machine à nombre fini d'états.

1. Les commandes asynchrones d'initialisation sont d'une utilité discutable dans le cas de fonctions complexes. Quand on utilise des circuits dont la polarité est programmable, l'optimiseur choisit la polarité qui conduit aux équations les plus simples. Les commandes asynchrones de forçage conduisent alors, dans certains de ces circuits, à un état inconnu, ce qui n'est, en principe, pas leur destination.

Le registre d'état est un registre synchrone de  $n$  éléments binaires. À chaque front d'horloge son contenu est remis à jour, l'état futur remplace l'état actuel.

La fonction combinatoire  $f()$  calcule l'état futur en fonction de l'état actuel et des entrées externes. Pour éviter tout risque d'aléa, ces entrées doivent être synchrones de l'horloge, si elles ne le sont pas naturellement, il convient de le prévoir lors de la conception de l'ensemble. Cette synchronisation des entrées est prévue dans la plupart des circuits complexes, indépendamment des blocs de calcul logiques. Pour la réaliser il suffit de créer un processus trivial qui associe à chaque entrée externe une bascule D.

La fonction combinatoire  $g()$  calcule les sorties en fonction de l'état actuel et, dans le cas d'une architecture de Mealy, des entrées. Sa complexité est liée au codage adopté pour les états. Quand cela est possible il est toujours préférable de la supprimer, en choisissant un codage du registre d'état adapté aux sorties désirées. Cette technique augmente généralement le nombre de bascules nécessaires, mais la logique combinatoire est plus simple. Globalement, le solde est souvent avantageux, et, surtout, les performances dynamiques du circuit sont très supérieures. Si la combinatoire de sortie est nécessaire, il est souvent souhaitable d'en synchroniser le résultat, là encore, de nombreux circuits offrent cette possibilité, sans pénaliser la capacité de calcul.

#### ► Du diagramme de transitions au programme

Une fois posé le synoptique général, sa traduction en un squelette de programme VHDL est immédiate :

```
entity proto_machine is
    generic (n , p , q : integer := 2 ) ;
    port(horloge : in bit ;
         entrees : in bit_vector(0 to p - 1);
         sorties : out bit_vector(0 to q - 1) ) ;
end proto_machine ;

architecture behave of proto_machine is
    signal etat_actuel : bit_vector(0 to n - 1) ;
begin

    machine_etats : process
        begin
            wait until horloge = '1' ;
            -- Ici prend place la description des transitions.
        end process machine_etats ;

    actions : process
        begin
            -- Ici prend place la description des sorties.
        end process actions ;
    end behave ;
```

Dans le prototype précédent, l'état interne du système est décrit par un signal de type structuré (`etat_actuel`), ici un vecteur d'éléments binaires, dont le pilote est créé par le processus `machine_etats` qui décrit bien un opérateur synchrone. La dimension de ce signal détermine le nombre de bascules du registre d'état. Bien que les compilateurs acceptent généralement l'utilisation de variables pour représenter l'état interne d'un système, nous déconseillons leur emploi pour modéliser des bascules physiquement présentes dans le circuit. Leur comportement temporel ne reproduit pas correctement celui des bascules. Nous aurons l'occasion de revenir sur ce point.

Pour décrire le fonctionnement dynamique une méthode classique consiste à établir un diagramme de transitions qui décrit l'évolution du système<sup>1</sup>, à chaque état de départ on associe des transitions qui se réalisent, au front actif d'horloge suivant, si les entrées vérifient une condition. Il existe des traducteurs de diagrammes de transitions, qui génèrent un programme source VHDL (ou d'autres langages) à partir de la représentation graphique du diagramme<sup>2</sup>. L'intérêt majeur de ces outils réside peut-être plus dans le contrôle de cohérence qu'ils exercent que dans la traduction elle-même.

Pour aller plus loin dans cette discussion nous allons traiter un exemple simple de plusieurs façons : un opérateur qui détecte dans une chaîne binaire d'entrée toutes les suites de trois zéros successifs figure 2-12.

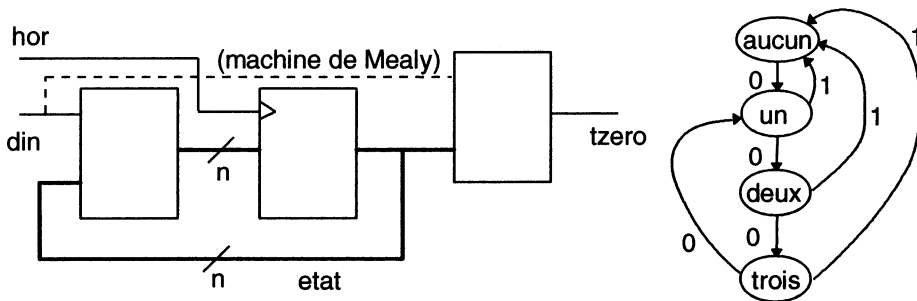


Figure 2-12 Détecteur de trois zéros successifs.

Quand le troisième zéro du flux d'entrée est détecté, la sortie `tzero` passe à un pendant une période d'horloge. Les données d'entrée sont synchrones de l'horloge.

1. Il ne faut pas être rigide, les diagrammes de transitions sont un outil visuel précieux pour décrire des automates simples dans lesquels les transitions n'obéissent pas à un algorithme précis. Il serait évidemment stupide de les utiliser pour décrire complètement un compteur à chargement parallèle, par exemple. Ce type de fonction se décrit beaucoup plus simplement par un algorithme qui tient en trois lignes. Il en est de même pour des opérateurs fondés sur les décalages, générateurs pseudo aléatoires et codeurs cycliques en tout genre.
2. State Cad, de DATA I/O, par exemple.

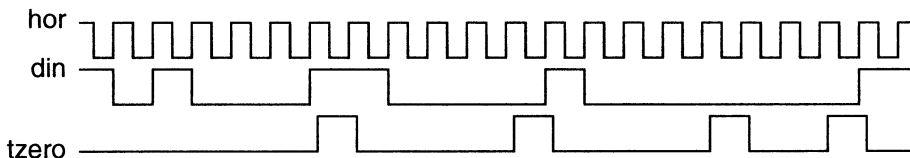
### » Machines de Moore

Dans une machine de Moore, les sorties sont calculées à partir de la valeur contenue dans le registre d'état. Le programme ci-dessous propose une solution qui correspond à cette architecture. L'état du système est matérialisé par un signal de type énuméré, dont il est inutile de préciser le codage :

```
entity trois_zero is
    port ( hor, din : in bit ;
          tzero : out bit ) ;
end trois_zero ;

architecture moore2 of trois_zero is
    type machine4 is (aucun, trois, un, deux) ;
    signal etat : machine4 ;
begin
    tzero <= '1' when etat = trois else '0' ;-- sortie
    std : process
    begin
        wait until hor = '1' ;
        if din = '1' then
            etat <= aucun ;
        else
            case etat is
                when aucun => etat <= un ;
                when un    => etat <= deux ;
                when deux  => etat <= trois ;
                when trois => etat <= un ;
            end case ;
        end if ;
    end process std ;
end moore2 ;
```

Un résultat de simulation du fonctionnement du programme est représenté figure 2-13. Il met en évidence une caractéristique des machines de Moore : la sortie est active à la période d'horloge qui suit l'événement détecté, ici le troisième zéro.



**Figure 2-13** Détection de trois zéros, machine de Moore.

Le simulateur utilisé est ici un simulateur postsynthèse de fichier JEDEC. Il indique un petit retard entre l'horloge et la sortie tzero. Ce retard, qui n'est pas lié au retard réel du circuit, assure un comportement causal du simulateur, c'est une forme simplifiée des cycles delta d'un simulateur VHDL.

### ➤ Codage des états

Un choix réfléchi du codage des états peut simplifier, ou même supprimer, le calcul des sorties. L'exemple précédent du détecteur de trois zéros peut être construit de sorte que la sortie tzero corresponde à l'état de l'une des bascules du registre d'états. Le synoptique et le diagramme de transitions de la figure 2-14 tiennent compte de ces remarques. Le code de chaque état est indiqué sur le diagramme de transitions.

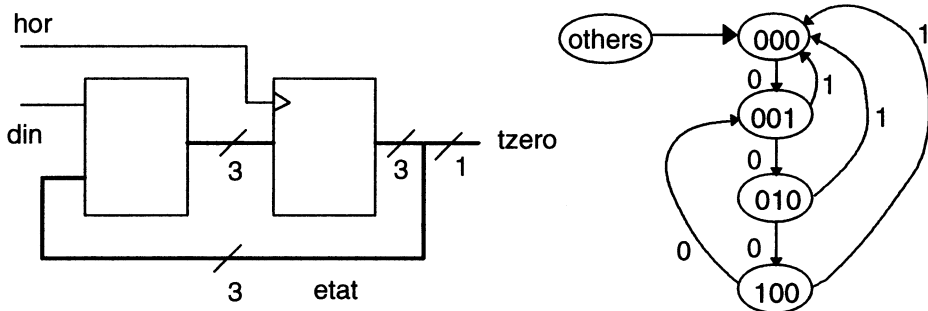


Figure 2-14 Détecteur de trois zéros, sortie directe.

Le codage utilisé comporte de nombreuses combinaisons inutilisées, puisque sur les huit états disponibles ( $2^3$ ) seuls quatre d'entre eux apparaissent dans le diagramme de transitions. Les états inutilisés doivent être raccordés dans le cycle normal de fonctionnement, c'est le sens de l'état générique nommé *others*.

Le programme VHDL ci-dessous utilise des constantes nommées pour décrire la nouvelle forme de l'architecture du détecteur de trois zéros ; la déclaration d'entité n'a pas subi de modification.

```
architecture moore3 of trois_zero is
    constant aucun      : bit_vector(2 downto 0) := "000" ;
    constant unzero     : bit_vector(2 downto 0) := "001" ;
    constant deuxzero   : bit_vector(2 downto 0) := "010" ;
    constant troiszero  : bit_vector(2 downto 0) := "100" ;
    signal etat : bit_vector(2 downto 0) ;
begin
    tzero <= etat(2) ;
    std : process
    begin
        wait until hor = '1' ;
        if din = '1' then
            etat <= aucun ;
        else
            case etat is
                when aucun      => etat <= unzero ;
                when unzero     => etat <= deuxzero ;
                when deuxzero   => etat <= troiszero ;
```



```

        when troiszero => etat <= unzero ;
        when others    => etat <= aucun ;
    end case ;
end if ;
end process std ;
end moore3 ;

```

Cette nouvelle solution nécessite une bascule de plus que la précédente, mais la suppression de la logique combinatoire de sortie et la grande simplicité des équations de commande des bascules compensent plus que largement cette apparente augmentation de complexité. L'utilisation directe des bascules du registre d'état, pour créer les sorties d'un système, offre de plus l'avantage non négligeable de réduire le temps de propagation entre le front actif de l'horloge et l'évolution des sorties.

Pour éviter à l'utilisateur d'avoir à décrire explicitement le codage des états, les outils de synthèse utilisent des attributs qui indiquent au compilateur la politique à suivre. Par exemple, dans le codage précédent, un état est repéré par une seule bascule dans l'état '1' ou toutes les bascules à '0'. Ce type de code est connu sous le nom de *one hot zero*, littéralement un seul élément binaire « chaud » ou aucun. Avec le compilateur WARP un tel code peut être généré à partir d'un type énuméré par les déclarations suivantes :

```

type machine4 is (aucun, trois, un, deux) ;
attribute state_encoding of machine4 :type is one_hot_zero ;
signal etat : machine4 ;

```

Cette méthode présente, malgré sa souplesse, des inconvénients :

- La portabilité des programmes n'est pas assurée, chaque compilateur utilise un jeu d'attributs qui lui est propre.
- Le raccordement des états inutilisés dans le diagramme de transitions n'est pas toujours garanti. Cette lacune est grave, si l'opérateur réalisé n'est pas initialisé correctement, il peut très bien parcourir un cycle parasite imprévu, ou se bloquer *ad vitam aeternam* dans un état imprévu. Il est alors indispensable de prévoir à la conception une entrée d'initialisation qui force la machine dans un état normal de son cycle.

### » Machines de Mealy

Dans une machine de Mealy, les entrées du système ont une action directe sur les sorties. Cette action directe permet de créer un mécanisme d'anticipation : une sortie de Mealy change de valeur avant que l'état de la machine n'entérine, éventuellement, la modification par une transition. Le monde extérieur est prévenu qu'une transition va avoir lieu au prochain front d'horloge. Le changement de valeur d'une sortie du type Moore indique, lui, qu'un changement d'état de la machine *vient de se produire*.

Le programme qui suit est une version « machine de Mealy » de notre détecteur de trois zéros. Nous avons repris la version à trois bascules précédente, en explicitant

l'état futur. Le calcul de la sortie est alors identique à celui de l'état futur trois-zero :

```
architecture mealy3 of trois_zero is
    constant aucun      : bit_vector(2 downto 0) := "000" ;
    constant unzero     : bit_vector(2 downto 0) := "001" ;
    constant deuxzero   : bit_vector(2 downto 0) := "010" ;
    constant troiszero  : bit_vector(2 downto 0) := "100" ;
    signal etat, futur : bit_vector(2 downto 0) ;
begin
    tzero <= futur(2) ;
    evolution : process
    begin
        wait until hor = '1' ;
        etat <= futur ;
    end process evolution ;

    transitions : process(din, etat)
    begin
        if din = '1' then
            futur <= aucun ;
        else
            case etat is
                when aucun      => futur <= unzero ;
                when unzero     => futur <= deuxzero ;
                when deuxzero   => futur <= troiszero ;
                when troiszero  => futur <= unzero ;
                when others     => futur <= aucun ;
            end case ;
        end if ;
    end process transitions ;
end mealy3 ;
```

Dans cette description, le processus *transitions*, qui calcule l'état futur en fonction de l'état actuel et de l'entrée *din*, représente une fonction logique combinatoire. Il est important de s'assurer de ce caractère combinatoire par une écriture minutieuse de l'algorithme associé, qui ne doit oublier aucune alternative des tests. Un oubli de cet ordre correspondrait à la création d'une mémoire de type *latch*.

Le résultat de simulation de la figure 2-15 confirme le mécanisme d'anticipation propre aux sorties de type Mealy : la sortie *troiszero* est active à la période d'horloge qui précède le changement d'état de la machine. Sur cette figure, issue du simulateur V-System, le vecteur qui représente l'état de la machine a été éclaté de façon à faire apparaître les valeurs de ses différentes composantes. Les stimuli sont les mêmes que ceux qui ont été utilisés pour la simulation de la machine de Moore (figure 2-13), de façon à illustrer le décalage d'une période d'horloge entre les deux solutions.

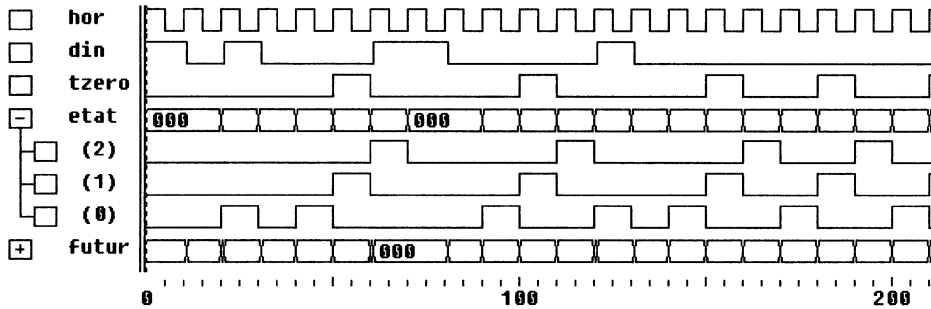


Figure 2-15 Détection de trois zéros, machine de Mealy.

### ► Moore ou Mealy ?

Le choix entre les deux architectures, Moore ou Mealy, est important à considérer dans une construction hiérarchique.

- Globalement, les solutions du type Moore sont plus simples à concevoir et à gérer : entièrement synchrones, elles peuvent être « emboîtées » les unes dans les autres, à la façon des poupées gigognes. Les rétrocouplages peuvent exister à tous les niveaux de la hiérarchie, sans risque de voir apparaître des aléas de fonctionnement dus à des rétroactions asynchrones.
- Quand le retard d'au moins une période d'horloge, inhérent à toute architecture de Moore, est incompatible avec les spécifications de fonctionnement, il peut être nécessaire d'envisager une architecture de Mealy, au moins pour certaines sorties. Dans un compteur décimal à plusieurs chiffres, comme dans l'exemple de la figure 2-8, chaque décade doit prévenir celle de poids immédiatement supérieur *avant* son passage à zéro, de façon à assurer une incrémentation synchrone et correcte de l'ensemble. La sortie dix de chaque décade est donc une sortie de Mealy : elle est active quand l'état de la décade vaut '9' et quand l'entrée d'autorisation, en, est active, condition qui indique que la décade *va* se remettre à zéro au prochain front d'horloge. Le programme ci-dessous est une solution possible.

```

library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all ;

entity decade is
  port (clk : in std_logic;
        en,raz : in std_logic;
        count: out std_logic_vector (3 downto 0);
        dix : out std_logic);
end decade ;

architecture behavior of decade is
  signal count_temp : integer range 0 to 9 ;
begin

```

```

count <= std_logic_vector ( -- simple conversion de type
    to_unsigned(count_temp,4) -- fonction de conversion1
    ) ;
dix <= en when count_temp = 9 else '0'; -- sortie Mealy
decimal : process
begin
    wait until rising_edge(ck) ;
    if (raz = '1') then
        count_temp <= 0 ;
    elsif (en = '1') then
        count_temp <= (count_temp + 1) mod 10 ;
    end if ;
end process decimal ;
end behavior ;

```

## 2.4 QUELQUES PIÈGES

VHDL est un langage concis, dont la sémantique est riche, que nous n'avons pas fini d'explorer. Avant de poursuivre notre étude du langage, il nous semble opportun de souligner quelques pièges classiques, dont la connaissance évitera au lecteur des recherches d'autant plus longues qu'elles sont entreprises tard, quand les programmes deviennent complexes.

Les pièges dangereux ne sont pas liés aux erreurs de syntaxe, celles-ci sont, en principe, détectées par les compilateurs ; c'est la moindre des choses<sup>2</sup>. Les pièges dangereux sont ceux qui proviennent d'une mauvaise compréhension du sens des choses, d'une perte de conscience de la structure du circuit décrit. Les pires des erreurs sont invisibles lors d'une simulation fonctionnelle superficielle, sont acceptées par les outils de synthèse et semblent même générer un circuit correct. Elles se manifestent de façon aléatoire, quand on teste le circuit réalisé au voisinage de ses limites de fonctionnement, près de sa fréquence maximum, par exemple.

Tous les exemples ci-dessous sont donc syntaxiquement corrects, et synthétisables. Ils ne produisent par contre pas la fonction annoncée, ou la produisent mal. Les défauts apparaissent évidemment lors d'un test du circuit réel, mais notre propos est d'indiquer comment éviter d'en arriver là. Les tests envisagés sont tous des tests en simulation, quand nous évoquons des tests postsynthèse, il s'agit de simulations des

1. On effectue ici une conversion en deux étapes : passage d'un entier à un type défini dans la librairie `numeric_std`, le type `unsigned`, qui est un vecteur d'objets du type `std_logic`, au moyen d'une fonction de cette librairie, puis conversion entre les vecteurs « proches », `unsigned` et `std_logic_vector`.

2. Le problème est le même en programmation, un programme doit toujours être présumé faux, les tests ayant pour but de mettre en évidence les erreurs. Un penchant naturel de tout programmeur est de tenter, au contraire, de se persuader de la justesse de son œuvre. Il organise les tests en conséquence, et arrive trop souvent à se convaincre. Un exemple pris au hasard : une fonction qui convertit un flottant en entier dont on omet de tester le comportement hors du domaine de définition du type cible !

modèles générés par les outils de conception. Ces modèles comportent les informations de retards internes des circuits.

### 2.4.1 Les mémoires cachées

La génération involontaire de mémoires de type *latch* est sans doute la faute la plus commune. À son origine on trouve le principe du pilotage par événements sous-entendu dans la sémantique des signaux : seuls les changements de valeurs des signaux ont à être indiqués dans un programme.

Quand on décrit le diagramme de transitions d'une machine d'états, seules les transitions doivent être spécifiées, le maintien éventuel est implicite. Cela est un avantage à condition que le fonctionnement de la machine soit décrit par un processus unique, synchrone de l'horloge, comme nous l'avons fait pour l'exemple précédent de la décade.

Si le diagramme de transitions comporte des maintiens, ce qui est souvent le cas, et que l'on adopte une architecture à deux processus, l'un pour le calcul des transitions, l'autre pour décrire le registre d'état, toute omission se traduit par un maintien asynchrone, comme le montre l'exemple suivant.

#### a) Un compteur à maintien asynchrone

Le programme ci-dessous modélise un simple compteur synchrone modulo 4, qui dispose d'une entrée d'autorisation de comptage (*en*) et d'une entrée de remise à zéro (*raz*) synchrones. Ces deux entrées sont actives au niveau '1', quand elles sont à '0', le compteur ne change pas d'état.

```
-- compteur modulo4, programme piège

entity compteur4 is
  port ( hor : in bit;
        en, raz : in bit ;
        etat: buffer integer range 0 to 3 );
end compteur4 ;

architecture mauvaise of compteur4 is
  signal futur : integer range 0 to 3 ;
begin
  transitions : process (raz,en,etat)
  begin
    if raz = '1' then
      futur <= 0 ;
    elsif en = '1' then
      futur <= (etat + 1) mod 4 ;
    end if ; -- mémoire de type latch par omission
  end process transitions ;
  evolution : process
  begin
    wait until hor = '1' ;
    etat <= futur ;
  end process evolution ;
end mauvaise ;
```

Si les signaux de commande du compteur sont synchrones de l'horloge, le fonctionnement semble correct. Les outils de synthèse ne protestent pas, et le circuit généré ne présente pas de caractère pathologique évident<sup>1</sup> : le compteur s'initialise convenablement si `raz = '1'`, quand ce dernier signal est inactif, le compteur s'incrémente normalement à chaque front d'horloge si `en = '1'`, reste dans son état initial si `en = '0'`. La figure 2-16 fournit un exemple de simulation trompeuse du circuit généré.

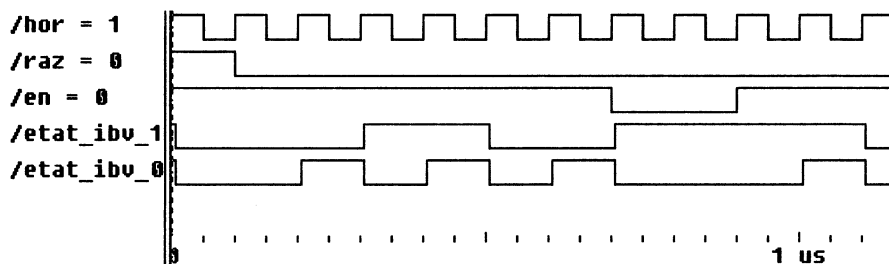


Figure 2-16 Une simulation rassurante (à tort).

#### b) Les indices de l'anomalie

Si on présente au même circuit des stimuli légèrement différents, le caractère asynchrone de son comportement apparaît. La figure 2-17 présente une trace de simulation pour laquelle l'entrée `en`, qui autorise ou inhibe le comptage, passe en position inhibition au milieu d'une période d'horloge. Contrairement à ce que l'on attend, le compteur s'incrémente d'une unité au front d'horloge suivant. L'origine de l'erreur est simple à comprendre : l'état mémorisé de façon asynchrone a eu « le temps » d'augmenter d'un, modulo 4, avant la mise en mémoire, d'où une discordance entre les contenus du registre d'état et de cette mémoire asynchrone.

Notons que la simulation fournit une image optimiste de ce qui se passerait réellement, l'état mémorisé n'a, en réalité, que très peu de chance d'être fixé par un

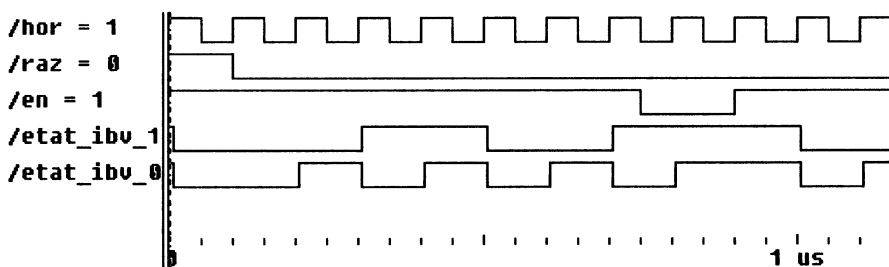


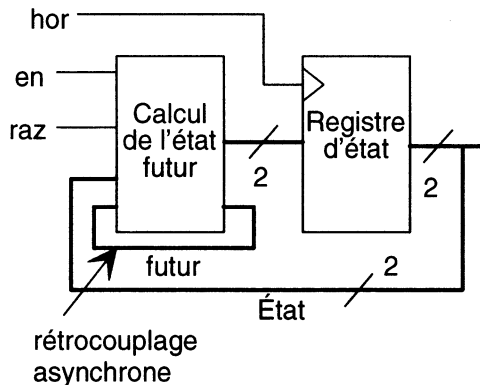
Figure 2-17 Où le maintien apparaît fantaisiste.

1. En simulation après synthèse, du moins. On sait que des mémoires asynchrones présentent des aléas qui peuvent, dans la réalité, conduire à des états aberrants lors du maintien dans l'état.

processus déterministe. Dans un système asynchrone le moindre écart entre les temps de propagation des opérateurs élémentaires du circuit conduit à une valeur mémorisée aléatoire.

Un deuxième indice de l'anomalie peut être observé en recherchant la fréquence maximum de travail du compteur précédent. Synthétisé dans un circuit de type PLD dont la fréquence maximum est de 40 MHz, par exemple, notre mauvaise version du compteur modulo 4 refuse de s'incrémenter correctement dès que la fréquence de l'horloge dépasse une vingtaine de mégahertz.

L'explication de cette baisse de performances devient claire quand on examine la structure des équations générées par le synthétiseur. L'architecture générale du circuit est celle de la figure 2-18 où nous avons indiqué le rétrocouplage asynchrone.



**Figure 2-18** Rétrocouplage asynchrone.

Les composantes du signal futur obéissent à des équations logiques qui contiennent ces mêmes composantes comme opérandes :

```
futur_IBV_1 = en * /raz * etat_IBV_0.Q * /etat_IBV_1.Q
              + en * /raz * /etat_IBV_0.Q * etat_IBV_1.Q
              + /en * /raz * futur_IBV_1

futur_IBV_0 = /en * /raz * futur_IBV_0
              + en * /raz * /etat_IBV_0.Q
```

Cette dépendance empêche l'optimisation en vitesse du circuit, optimisation qui consisterait à supprimer la séparation en deux blocs séparés des équations de la machine d'états.

### c) Les remèdes

La première façon d'éviter la création de bascules asynchrones pour mémoriser l'état d'un système consiste à ne pas séparer en deux processus distincts la description d'une machine d'états. Dans le programme ci-dessous le maintien dans l'état est assuré de façon synchrone, quelles que soient les alternatives des différents tests :

```

architecture un_process of compteur4 is
begin
  evolution : process
  begin
    wait until hor = '1' ;
    if raz = '1' then
      etat <= 0 ;
    elsif en = '1' then
      etat <= (etat + 1) mod 4 ;
    end if ; -- mémoire synchrone
  end process evolution ;
end un_process ;

```

Si l'on souhaite maintenir, malgré tout, le découpage en deux processus de la description de la machine d'états, il est essentiel d'assurer le caractère combinatoire du processus qui calcule les transitions.

Une première méthode consiste à prévoir une alternative par défaut dans les tests :

```

architecture correcte of compteur4 is
  signal futur : integer range 0 to 3 ;
begin
  transitions : process (raz,en,etat)
  begin
    if raz = '1' then
      futur <= 0 ;
    elsif en = '1' then
      futur <= (etat + 1) mod 4 ;
    else
      futur <= etat ; -- défaut : maintien dans l'état
    end if ;
  end process transitions ;
  evolution : process
  begin
    wait until hor = '1' ;
    etat <= futur ;
  end process evolution ;
end correcte ;

```

Une seconde politique consiste à prévoir au début de tout processus qui représente un opérateur combinatoire une « initialisation » explicite des signaux qui sont l'objet d'une affectation dans le processus. Cette valeur par défaut n'a aucun effet si elle est remplacée, compte tenu des résultats des tests, par une autre :

```

architecture val_init of compteur4 is
  signal futur : integer range 0 to 3 ;
begin
  transitions : process (raz,en,etat)
  begin
    futur <= etat ; -- défaut : maintien dans l'état
    if raz = '1' then
      futur <= 0 ;
    end if ;
  end process transitions ;
  evolution : process
  begin
    wait until hor = '1' ;
    etat <= futur ;
  end process evolution ;
end val_init ;

```



```

        elsif en = '1' then
            futur <= (etat + 1) mod 4 ;
        end if ;
    end process transitions ;
    evolution : process
    begin
        wait until hor = '1' ;
        etat <= futur ;
    end process evolution ;
end val_init ;

```

Lors de la synthèse le signal futur est supprimé par l'optimisation, ce qui conduit à des équations identiques pour nos trois dernières versions :

```

etat_IBV_0.D = en * /raz * /etat_IBV_0.Q
              + /en * /raz * etat_IBV_0.Q

etat_IBV_0.C = hor

etat_IBV_1.D = en * /raz * etat_IBV_0.Q * /etat_IBV_1.Q
              + /raz * /etat_IBV_0.Q * etat_IBV_1.Q
              + /en * /raz * etat_IBV_1.Q

etat_IBV_1.C = hor

```

La version corrigée du compteur accepte, c'est heureux étant donnée la simplicité de la fonction réalisée, une fréquence d'horloge égale à celle indiquée pour le circuit utilisé, à savoir 40 MHz.

## 2.4.2 Signaux et variables

Un signal représente une équipotentielle du monde physique, éventuellement assortie d'une cellule mémoire ; sa valeur est mise à jour quand le processus qui le pilote se met en attente. Les variables ne sont que des éléments de stockage temporaires utiles dans certains algorithmes. La différence majeure de comportement entre un signal et une variable est que le premier conserve sa valeur du début à la fin d'un algorithme séquentiel, alors que la seconde est actualisée immédiatement par une instruction d'affectation.

### a) Un générateur de parité fantaisiste

La différence de comportement est évidente si on considère une version fautive du générateur de parité dont nous avons donné précédemment une version correcte, qui utilise une variable<sup>1</sup> :

```

ENTITY ouex IS
    generic(taille : integer := 8) ;
    PORT ( a : IN BIT_VECTOR(0 TO taille - 1) ;
          s : OUT BIT );

```

1. Voir paragraphe 2.3.2, présentation des boucles.

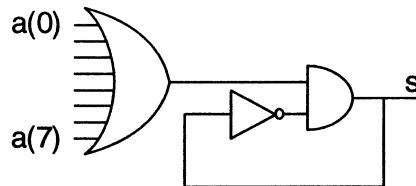
```

END ouex;

ARCHITECTURE fausse of ouex is
    signal parite : bit ; -- MAUVAIS CHOIX
BEGIN
    process(a)
    begin
        parite <= '0' ;
        FOR i in a'range LOOP
            if a(i) = '1' then
                parite <= not parite;
            end if;
        END LOOP;
        s <= parite;
    end process;
END fausse ;

```

La différence entre ce programme et le précédent réside en l'utilisation d'un signal *parite*, au lieu d'une variable de même nom interne au processus. Ce signal prendra la valeur '0' si aucune des composantes du vecteur d'entrée ne vaut '1', et sera complémenté si l'une au moins de ces composantes vaut '1', ce qui n'a pas grand rapport avec la fonction souhaitée, comme l'atteste le schéma de la figure 2-19 que traduit ce programme.



**Figure 2-19** Un curieux générateur de parité.

Le programme précédent, bien que faux, est parfaitement synthétisable. Le circuit généré a un comportement pathologique : il contient une réaction interne, sur un opérateur combinatoire inverseur. Si l'une des entrées est différente de '0', la sortie oscille à une fréquence qui dépend des temps de propagation dans le circuit cible.

#### *b) Variables et bascules*

Bien que leur comportement vis-à-vis du temps ne corresponde pas à celui des bascules du monde physique, les variables conservent leurs valeurs d'une activation à l'autre d'un processus. Elles peuvent donc mémoriser l'état interne d'un système séquentiel. Cette utilisation des variables conduit certains compilateurs, lors de la synthèse, à les transformer en signaux, quitte à modifier en conséquence les équations de façon à retrouver le même comportement « externe » pour l'opérateur.

L'exemple qui suit illustre ce point. On y modélise un diviseur de fréquence par dix : un processus utilise une variable interne en compteur par 5, et complémente la valeur du signal de sortie à chaque nouveau cycle de comptage.

```
entity div_10 is
  port( hor : in bit ;
        sort : buffer bit );
end div_10 ;

architecture horrible of div_10 is -- A éviter
begin
  diviseur : process
    variable compte : integer range 0 to 5 := 0 ;
  begin
    wait until hor = '1' ;
    compte := compte + 1 ;
    if compte = 5 then
      compte := 0 ;
      sort <= not sort ;
    end if ;
  end process diviseur ;
end horrible ;
```

Ce programme, bien qu'effectuant la fonction annoncée, concentre en quelques lignes ce qu'il ne faut pas faire avec une variable :

- la variable `compte` prend inutilement les valeurs de 0 à 5 ;
- quand elle atteint la valeur 5 elle est précipitamment remise à zéro, la valeur 5 ne dure donc qu'un temps nul ;
- lors de la synthèse elle est remplacée par un signal qui compte de 0 à 4, comme il se doit. Les valeurs de la variable et du signal qui la remplace ne sont donc pas les mêmes, ce qui ne facilite guère la recherche d'éventuelles erreurs.

Outre les maladrresses d'écriture qui rendent le code source peu lisible, l'auteur d'un tel programme prend le risque de se heurter à des difficultés de portabilité, tous les compilateurs ne traitent pas les variables de la même façon lors de la synthèse<sup>1</sup>.

Nous préférons de loin la version suivante du même diviseur par dix, version dans laquelle le registre d'état est matérialisé par un signal :

```
architecture correcte of div_10 is
  signal compte : integer range 0 to 4 := 0 ;
begin
  diviseur : process
  begin
    wait until hor = '1' ;
    if compte = 4 then
      compte <= 0 ;
      sort <= not sort ;
    end if ;
  end process diviseur ;
end correcte ;
```

1. À notre sens la prudence élémentaire consisterait à refuser la synthèse de tels programmes, mais ce n'est pas le cas.

```

        else
            compte <= compte + 1 ;
        end if ;
    end process diviseur ;
end correcte ;

```

Ou, si l'outil de synthèse accepte les opérations de division (ici l'opérateur mod) par des nombres qui ne sont pas des puissances de deux :

```

architecture modulo5 of div_10 is
    signal compte : integer range 0 to 4 := 0 ;
begin
    diviseur : process
    begin
        wait until hor = '1' ;
        compte <= (compte + 1) mod 5 ;
        if compte = 4 then
            sort <= not sort ;
        end if ;
    end process diviseur ;
end modulo5 ;

```

### 2.4.3 Les boucles

L'habitude de la programmation des microprocesseurs peut inciter à utiliser les instructions de boucles pour gérer le temps. Quoi de plus naturel, semble-t-il, que de créer un compteur, par exemple, au moyen d'une structure de boucle ? Il n'en est rien. Rappelons que la boucle fondamentale qui permet de gérer le temps est celle que constitue le corps d'un processus lui-même ; les instructions de boucles servent à créer des structures spatiales répétitives, à décrire par des algorithmes des opérations qui peuvent très bien être « instantanées ». Les deux exemples qui suivent sont syntaxiquement corrects et fournissent les résultats attendus en simulation. Le premier est généralement refusé lors de la synthèse du circuit, le second qui n'est pas synthétisable, et ne cherche pas à l'être, correspond à du code inutile : celui de la boucle explicite utilisée.

#### a) Des boucles non synthétisables

Une autre version du diviseur de fréquence précédent est la suivante :

```

architecture boucle of div_10 is -- non synthétisable
begin
    diviseur : process
    begin
        for compte in 0 to 4 loop
            wait until hor = '1' ; -- suspension dans la boucle
            if compte = 4 then
                sort <= not sort ;
            end if ;
        end loop ;
    end process diviseur ;
end boucle ;

```

D'un fonctionnement sans histoire en simulation, cette version du diviseur pose un problème lors de la synthèse. Pour reproduire le fonctionnement souhaité, le compilateur devrait transformer le compteur de boucle (`compte`) en un signal mémorisé dans des bascules ; opération encore plus scabreuse que la transformation d'une variable en un signal. Cette construction est donc refusée en synthèse.

Une comparaison attentive des architectures « `modulo5` » et « `boucle` », du diviseur de fréquence, montre qu'en réalité la boucle est aussi inutile que non synthétisable. La version synthétisable de la fonction a le mérite de mettre en évidence les signaux mémorisés, sans être plus compliquée, bien au contraire.

### *b) Des boucles inutiles*

Autre variation sur le thème des boucles inutiles : la construction d'un générateur de signaux à des fins de test.

On se propose de créer un composant qui génère, en simulation, des impulsions périodiques de période 40 ns. Un tel objet virtuel est utile pour créer, par exemple, un signal d'horloge dans un programme de test. Une première version, juste mais naïve, est :

```
entity hor_simple is
    port (hor : buffer bit );
end hor_simple ;

architecture naive of hor_simple is
begin
    horloge : process
    begin
        hor <= '0' ;
        loop
            wait for 20 ns ;
            hor <= not hor ;
        end loop ;
    end process horloge ;
end naive ;
```

Si l'on remarque que la valeur initiale d'un signal binaire est '0', et qu'un processus est en lui-même une boucle sans fin, on obtient le programme équivalent :

```
architecture simple of hor_simple is
begin
    horloge : process
    begin
        wait for 20 ns ;
        hor <= not hor ;
    end process horloge ;
end simple ;
```

Qui fournit le même résultat, et dans lequel toute instruction de boucle a disparu.

En réalité, l'instruction « wait » elle-même peut être éliminée de notre générateur d'horloge ; il suffit d'indiquer au processus de créer un événement futur et d'attendre cet événement :

```
architecture minimum of hor_simple is
begin
  horloge : process (hor)
  begin
    hor <= not hor after 20 ns ;
  end process horloge ;
end minimum ;
```

Résumons, au risque de nous répéter :

- L'évolution dans le temps est une boucle sans fin implicite. Boucle dont le déroulement est contrôlé par la création et l'attente d'événements. Les signaux sont les objets qui permettent la communication des événements entre les processus.
- Les boucles servent à construire des algorithmes. Dans un programme synthétisable toute boucle commencée doit arriver à son terme dans le même instant. Elle ne contient donc aucune dimension temporelle. C'est dans ce sens que nous les avons qualifiées de spatiales.
- Plus généralement, on se limitera, dans les programmes synthétisables, à des structures de boucles aussi simples que possibles, dans lesquelles le nombre d'itérations effectuées est connu à la compilation. L'exemple typique de l'utilisation d'une boucle est le traitement de tous les éléments d'un tableau.

## 2.4.4 La complexité sous-jacente

Lors du processus de synthèse, une description est traduite en équations logiques qui ne font intervenir que des opérateurs élémentaires (portes logiques, multiplexeurs à deux ou quatre entrées, bascules D, T ou JK). Une expression apparemment simple, dans un langage évolué comme VHDL, peut engendrer des équations extrêmement complexes, voire irréalisables dans la technologie du circuit cible.

La première précaution est, évidemment, de proscrire l'utilisation de nombres dont le domaine de variation n'est pas spécifié, de limiter les tailles des tableaux aux valeurs minimums acceptables, de contrôler que le nombre de bascules n'excède pas celui prévu<sup>1</sup>.

Au-delà de ces pièges élémentaires, certaines descriptions peuvent conduire à des difficultés lors de la synthèse. Le premier exemple que nous citerons concerne les opérations arithmétiques, le second la gestion des signaux d'horloge.

1. Tous les compilateurs fournissent dans leur compte rendu le nombre de bascules, *edge* et *latch*, impliquées par le code source.

### a) Les opérations arithmétiques ou l'explosion combinatoire

La synthèse directe d'un opérateur arithmétique génère un circuit combinatoire. Les fonctions logiques qui réalisent ces opérations comportent très peu de simplifications<sup>1</sup>, ce qui rend inapplicable, dès que les dimensions des opérandes dépassent quelques unités, leur réalisation en trois couches logiques (première ou seconde forme canonique).

À titre d'exemple, un multiplieur de deux nombres compris entre 0 et 7 (3 bits) nécessite 35 produits logiques allant jusqu'à 6 (2 fois 3) variables. Quand on passe à des opérandes compris entre 0 et 15 (4 bits), le nombre de produits passe à 144. Si on tente de poursuivre, on arrive à 553 produits logiques regroupés par des ou qui nécessitent jusqu'à 132 entrées pour un multiplieur dont les opérandes sont compris entre 0 et 31. Sans compter le temps de calcul nécessaire à la réduction des dix fonctions de dix variables obtenues.

Un tel monstre est obtenu, moyennant un pilotage catastrophique (l'exigence de générer un schéma en deux couches logiques) du compilateur, à partir d'un programme aussi simple que :

```
entity multi is
  generic (max : integer := 31);
  port ( a, b : in integer range 0 to max;
        prod : out integer range 0 to max*max );
end multi;

architecture brute of multi is
begin
  prod <= a * b;
end brute;
```

Les solutions au problème de la complexité des opérations arithmétiques dépendent de l'application envisagée. Nous nous contenterons de donner quelques indications :

- N'y a-t-il pas d'autre solution que de passer par des opérations arithmétiques générales ?
- Le problème peut-il être traité de façon séquentielle ? Les opérations arithmétiques ont des solutions très simples, dont la complexité dépend peu (multiplications) ou pas du tout (additions) de la dimension des opérandes, si elles sont effectuées en « série ». Ces solutions sont notamment bien adaptées aux problèmes des filtres numériques qui gèrent des flots de données. Dans ce cas, le traitement en série permet d'accroître la vitesse de traitement en terme de flux, au prix d'un retard constant. C'est le principe des architectures *pipe line*.

1. Tous les compilateurs comportent un module de simplification logique. Les algorithmes utilisés sont des classiques du genre : *Espresso* ou *Quine-McCluskey*. Le premier, contrairement au second, ne garantit pas une forme minimale, mais évite l'examen de toutes les combinaisons des variables d'entrée, ce qui le rend applicable à des fonctions d'un grand nombre de variables.

- Étudier l'influence des paramètres de l'optimiseur (le *fitter*). Ces paramètres permettent de contrôler le nombre de couches logiques générées. Quand on augmente le nombre de couches, la complexité diminue mais le temps de calcul augmente.
- Utiliser les modules arithmétiques des bibliothèques « fondeur ». Tous les fabricants de circuits programmables complexes fournissent la possibilité d'instancier des modules arithmétiques, optimisés en fonction de la technologie des circuits. Le défaut majeur de cette solution est qu'elle nuit à la portabilité des programmes. Certains compilateurs tentent de rendre transparente cette instanciation en l'inférant des opérateurs arithmétiques eux-mêmes.

### b) Les horloges multiples

Terminons ce tour d'horizon de quelques pièges par une « perle » synthétisable :

```
entity horbad is
  port ( vechor : in bit_vector(2 downto 0);
        din  : in bit_vector(7 downto 0);
        dout : out bit_vector(7 downto 0) );
end horbad;

architecture bizarre of horbad is
begin
  mem : process
  begin
    wait until vechor = "111";
    dout <= din;
  end process mem;
end bizarre;
```

Un circuit synchrone dont l'horloge est un vecteur ! Le résultat de la synthèse, qui traduit correctement le sens du programme, comporte un opérateur ET sur le signal d'horloge, conformément au schéma de la figure 2-20.

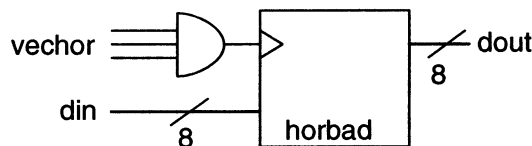


Figure 2-20 Une horloge dangereuse.

Impossible à implémenter dans la plupart des circuits programmables, dont les signaux d'horloge sont prédéfinis et dépourvus de tout opérateur combinatoire, une telle construction peut être introduite dans un ASIC. Contraires à toutes les règles de conception, de tels montages génèrent des décalages temporels incontrôlables (*clock skew*) entre les fronts d'horloges qui commandent les opérateurs séquentiels d'un circuit. Cette perte de synchronisme peut provoquer des violations de temps de maintien à l'origine d'erreurs de fonctionnement difficiles à localiser.



## 2.5 PROGRAMMATION MODULAIRE

VHDL, comme tous les langages évolués, permet de subdiviser un algorithme complexe en modules plus simples, compilables et testables séparément, que l'on peut ranger dans des bibliothèques et importer à volonté.

VHDL décrit des circuits, des architectures matérielles. Un circuit complexe peut être obtenu en assemblant des blocs fonctionnels plus simples, blocs internes à une architecture ou composants, pour lesquels des politiques différenciées de synthèse (placement et routage, compromis vitesse-surface lors de l'optimisation) peuvent être établies.

Le langage offre donc deux outils complémentaires de constructions modulaires :

- un outil purement logiciel, qui passe par la constitution de sous-programmes, de paquetages et de bibliothèques,
- un outil matériel, nommé traditionnellement construction hiérarchique.

Le premier ne laisse pas de trace dans le silicium nous le décrirons en premier, le second peut survivre au processus de synthèse et laisser sa marque dans le circuit final ; nous en avons déjà décrit certains aspects, le prochain paragraphe lui est consacré.

### 2.5.1 Les sous-programmes : procédures et fonctions

VHDL connaît deux catégories de sous-programmes : les procédures et les fonctions. Les sous-programmes sont des modules de code séquentiel<sup>1</sup> ; ils utilisent donc le même jeu d'instructions que les processus, sauf l'instruction `wait` qui est sujette à des restrictions. En simulation cette instruction est utilisable sous certaines réserves, en synthèse elle est généralement interdite.

La création et l'emploi d'un sous-programme passent par trois opérations distinctes :

- Sa création proprement dite, ou définition, consiste à écrire l'algorithme qui constitue le corps du sous-programme.
- Sa déclaration consiste à rendre connu d'un module qui l'utilise le prototype du sous-programme, c'est-à-dire son nom, ses arguments d'appels et, dans le cas d'une fonction, le type de la valeur retournée.
- Son utilisation dans une instruction établit un lien entre les paramètres formels, utilisés dans la définition et la déclaration, et les paramètres réels qui appartiennent au module appelant.

Déclaration et définition constituent du code passif, elles figurent dans la zone déclarative d'une entité, d'une architecture, d'un bloc, d'un processus, d'un autre sous-programme, ou, de préférence dans un paquetage qui peut être compilé séparément. Si, ce qui n'est pas particulièrement une bonne pratique de programmation, un

---

1. Séquentiel est pris ici au sens langage de programmation. Cela n'implique en rien un circuit séquentiel.

sous-programme est défini dans la zone déclarative du module qui l'utilise, il est inutile de rajouter une déclaration qui serait alors redondante.

En synthèse, l'emploi de sous-programmes ne fait évidemment faire aucune « économie de silicium », chaque appel à une procédure ou à une fonction provoque la création des opérateurs physiques nécessaires à sa réalisation. On peut rapprocher ce mécanisme de celui des fonctions *in line* de certains langages de programmation.

Procédures et fonctions diffèrent principalement par les sens de transfert des informations, alors que les premières autorisent des paramètres en entrée et en sortie, les secondes n'ont que de paramètres qu'en entrée, et renvoient un résultat utilisable dans une expression (figure 2-21).

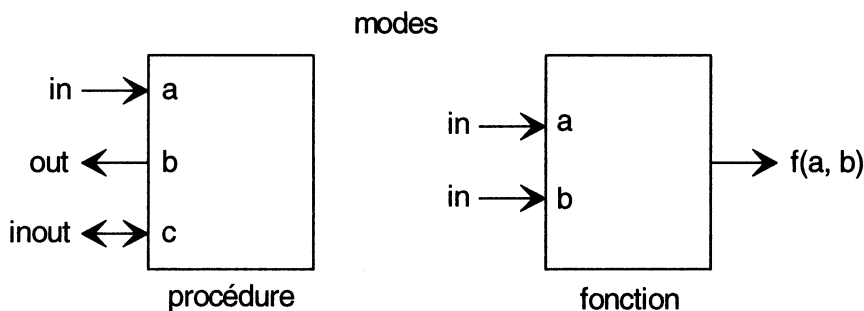


Figure 2-21 Procédures et fonctions.

### a) Syntaxe

Les sous-programmes sont des « composants » logiciels. Il n'est donc guère étonnant de retrouver à propos de leur usage des constructions syntaxiques semblables à celles qui ont été explorées à propos de l'utilisation des composants, notamment les listes d'association.

#### ► Définition

La définition du corps d'un sous-programme comporte, comme à l'habitude, deux parties : une partie déclarative où sont définis les objets (données, types ...) locaux et le corps proprement dit qui contient une suite d'instructions séquentielles. Les sous-programmes faisant partie du monde séquentiel, leurs données locales sont des constantes, des variables ou, en simulation, des fichiers. La zone déclarative peut également contenir les définitions de sous-programmes internes, emboîtés dans celui que l'on est entrain de décrire<sup>1</sup>.

L'instruction `wait` est interdite dans le corps d'une fonction, elle peut figurer dans celui d'une procédure ; mais cette possibilité est souvent exclue par les compilateurs

1. On relèvera qu'il y a là une différence notable avec le langage C pour lequel les fonctions sont toujours des objets externes.

en synthèse. Une fonction a un type qui correspond à la valeur retournée grâce à l'instruction `return`. La syntaxe de définition d'un sous-programme est :

```
subprogram_body ::=
  procedure nom [(liste_des_paramètres_formels)]
| function nom [(liste_des_paramètres_formels)] return type
is
  zone déclarative
begin
  zone d'instructions séquentielles
end [procedure | function] [nom];
```

Les paramètres formels ont une classe, un mode et un type :

```
liste_des_paramètres_formels ::=
  interface_element {, interface_element }
interface_element ::=
  [classe] liste_de_noms : [mode] sous_type [:= expression]
```

Procédures et fonctions diffèrent sur ce point :

- Les paramètres d'une procédure peuvent être de classe `signal`, `constant`, `variable` ou `file`. Pour les trois premières classes, les modes acceptés sont `in`, `out` et `inout`, les fichiers n'ont pas de mode.
- Les paramètres d'une fonction peuvent être des constantes ou des signaux, toujours de mode `in`, ou des fichiers ; jamais des variables.
- Classes et modes peuvent être omis avec les conventions suivantes :
- Le mode par défaut est `in`.
- Un paramètre de mode `in` est, par défaut, une constante.
- Un paramètre de mode `out` ou `inout` est, par défaut, une variable.

Une expression statique<sup>1</sup> permet de fixer la valeur par défaut d'un paramètre de mode `in`. Ce paramètre peut, à cette condition, être omis lors de l'appel du sous-programme.

Sans entrer dans les détails, soulignons qu'un sous-programme qui reçoit en paramètre un objet structuré, tableau ou enregistrement, a toutes les informations nécessaires à la manipulation de cet objet ; dimensions, bornes des indices, etc. Si le paramètre formel est un tableau non contraint, les dimensions sont déterminées au moment de l'appel. Cette souplesse permet de créer des bibliothèques générales indépendantes des dimensions des objets manipulés.

Les variables locales d'un sous-programme sont dynamiques : elles n'existent que quand le sous-programme est actif et sont réinitialisées à chaque appel. Cela implique que si le sous-programme ne contient pas d'instruction `wait`, ce qui est toujours le cas des fonctions, les variables locales ont une durée de vie nulle au sens du temps simulateur. En synthèse elles ne peuvent donc en aucun cas générer de cellule mémoire<sup>2</sup>. Ce comportement des variables d'un sous-programme est

1. Une expression statique définit une valeur connue au moment de l'analyse du programme, par opposition à une expression dont la valeur change au cours de l'exécution (simulation) du programme.

2. Heureusement ! Nous excluons ici les variables partagées, mentionnées précédemment.

différent de celui des variables d'un processus qui a, comme ses variables, une durée de vie éternelle.

Le programme qui suit illustre cette différence de comportement. Une fonction, une procédure et deux processus contiennent des algorithmes similaires, qui utilisent une variable. Les deux premiers fournissent à chaque appel des résultats indépendants des appels précédents; le processus mémoire se « souvient » de ses activations passées alors que le processus oubli réinitialise sa variable à chaque activation, et fournit donc le même résultat que la procédure et la fonction (figure 2-22).

```
entity dynam is end dynam;1

architecture simple of dynam is
    signal stim, f_ssprg, p_ssprg,
           procmem, procoubli : integer := 0 ;

    -- définition de la fonction :
    function essf (signal sin : integer) return integer is
        variable coeff : integer := 2 ;
    begin
        coeff := coeff * 2 ;
        return sin + coeff ;
    end function essf ;

    -- définition de la procédure :
    procedure essp (signal sin : in integer ;
                  signal sort : out integer) is
        variable coeff : integer := 2 ;
    begin
        coeff := coeff * 2 ;
        sort <= sin + coeff ;
    end procedure essp ;

begin
    stim    <= 6 after 10 ns, 0 after 20 ns ;
    f_ssprg <= essf(stim) ; -- appel de la fonction
    essp(stim, p_ssprg) ; -- appel de la procédure

    memoire : process ( stim )
        variable coeff : integer := 2 ;
    begin
        coeff := coeff * 2 ;
        procmem <= stim + coeff ;
    end process memoire ;

    oubli : process ( stim )
        variable coeff : integer ;
```

1. Un tel « circuit » qui n'a ni entrée ni sortie est un pur module de test, qui n'a de sens qu'en simulation.

```

begin
    coeff := 2 ;
    coeff := coeff * 2 ;
    procoubli <= stim + coeff ;
end process oubli ;
end simple ;

```

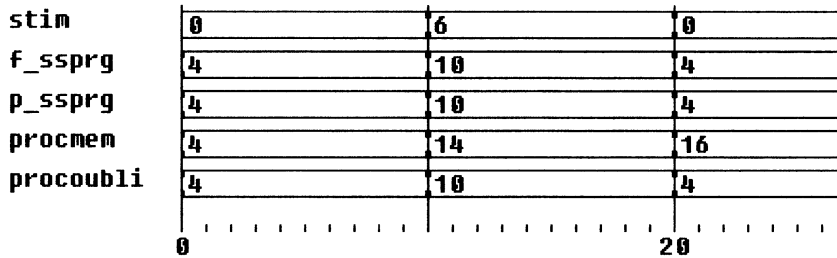


Figure 2-22 Mémoire des variables locales.

Il apparaît bien que seule la variable du processus mémoire est multipliée par deux à chaque activation.

### » Déclaration

Dans l'exemple précédent, fonctions et procédures n'étaient pas déclarées, leur définition servait de déclaration. Quand la référence à un sous-programme survient avant sa définition dans un fichier source, il est nécessaire de prévoir une déclaration préliminaire. Ce genre de situation se présente essentiellement quand on construit le code d'une librairie. La syntaxe de la déclaration reprend l'en-tête de la définition :

```

subprogram_declaration ::=
    procedure nom [(liste_des_paramètres_formels)]
    | function nom [(liste_des_paramètres_formels)] return type ;

```

### » Appel

L'appel d'une procédure est une instruction à part entière, concurrente ou séquentielle. L'appel d'une fonction intervient comme opérande (valeur retournée) dans une expression quelconque. L'association<sup>1</sup> entre les paramètres réels (qui appartiennent

1. Le manuel de référence définit deux catégories de constructions à première vue voisines, mais dont les syntaxes et les significations diffèrent :

- a. *association\_element* : élément d'association, intervient dans les appels de sous-programmes et les instructions `port map`. Il s'agit d'une interface élémentaire, un lien, entre deux niveaux hiérarchiques. Ces associations peuvent éventuellement être coupées (*open*).
- b. *element\_association* : association entre éléments, intervient dans les choix d'objets à associer à d'autres, comme dans les agrégats. Ces associations autorisent des groupements multiples (*range*, *others*).

au programme appelant) et les paramètres formels (internes au sous-programme) peut se faire par position ou par association nommée explicite :

```
function_call ::=
    function_name[(association_list)]
procedure_call_statement ::=
    [label :] procedure_name[(association_list)] ;
```

Où :

```
association_list ::=
    association_element{, association_element }
association_element ::= [formal_part =>] actual_part
```

Comme pour l'association de ports ou de paramètres génériques, lors de l'instanciation d'un composant, si une liste d'association contient à la fois des associations par positions et des associations nommées, les premières doivent précéder les secondes dans le bon ordre.

Il est possible d'omettre une association si le paramètre formel correspondant est de mode in, de classe variable ou constante et si une valeur par défaut a été définie dans le sous-programme.

### ► Exemple

L'exemple qui suit reprend la modélisation d'un contrôleur de parité dont une première version a été fournie précédemment, comme illustration des boucles. Cet exemple nous permet de souligner la souplesse de manipulation des tableaux passés en arguments à des sous-programmes : nous souhaitons rendre le contrôleur de parité indépendant de la taille du vecteur dont on calcule la parité, taille fixée par un paramètre générique de l'entité, donc inconnue lors de l'écriture du sous-programme. Les attributs se rapportant à des tableaux permettent de retrouver les plages de variation des indices de façon dynamique :

```
ENTITY ouex IS
    generic(taille : integer := 8) ;
    PORT ( a : IN BIT_VECTOR(0 TO taille - 1) ;
           s : OUT BIT );
END ouex;

ARCHITECTURE proc of ouex is
    procedure pariproc (signal e : in bit_vector ; signal
                        impair : out bit ) is
        variable parite : bit := '0' ;
    begin
        FOR i in e'range LOOP -- taille quelconque
            if e(i) = '1' then
                parite := not parite;
            end if;
        END LOOP;
        impair <= parite;
    end procedure pariproc ;
```

```

BEGIN
    pariproc(a,s) ; -- appel de la procédure
    -- on aurait pu écrire pariproc(impair => s,e => a) ;
    -- ou encore          pariproc(e => a, impair => s) ;
END proc ;

```

Ou, avec une fonction :

```

ARCHITECTURE fonct of ouex is

    function parifonc (signal e : in bit_vector ) return bit is
        variable parite : bit := '0' ;
    begin
        FOR i in e'range LOOP
            if e(i) = '1' then
                parite := not parite;
            end if;
        END LOOP;
        return parite;
    end function ;

BEGIN
    s <= parifonc(a) ;
END fonct ;

```

La lourdeur de la zone déclarative des deux architectures précédentes disparaîtra quand nous apprendrons à créer des paquetages dans lesquels tous les sous-programmes nécessaires à une application trouveront leur place.

### b) Surcharges de sous-programmes

La notion de surcharge (ou surdéfinition, *overloading*) est apparue au début des années 80 avec ADA et les langages objets. Elle permet de donner le même nom à des sous-programmes ayant une action similaire sur des types de données différents, sans interdire au compilateur d'exercer un contrôle strict sur les types des expressions<sup>1</sup>.

#### ➤ Des actions adaptées aux types des arguments

Considérons le cas simple des opérateurs arithmétiques, l'addition, par exemple.

Les formats binaires des nombres entiers et des nombres flottants sont très différents, les premiers sont simplement les coefficients du développement du nombre en base deux, les seconds contiennent des champs d'exposant et de mantisse, assortis de conventions d'écriture que nous ne détaillerons pas ici. Derrière le même symbole opératoire '+' se cachent des opérations fort différentes. La nature de ces

1. Le problème est contourné plus que résolu en C par les fonctions à nombre variable d'arguments comme printf(...) et scanf(...). Tout programmeur connaît les erreurs d'exécutions occasionnées par ces fonctions auxquelles tout contrôle de cohérence des types a été retiré.

opérations dépend des types des opérandes, c'est-à-dire des types d'arguments passés à un sous-programme.

Tous les langages possèdent en interne la notion de surcharge, ne serait-ce que pour les opérations arithmétiques, mais elle est implicite. Tous ne permettent pas de créer explicitement des programmes surchargés<sup>1</sup>.

En VHDL un sous-programme est déterminé de façon univoque par son nom et sa signature :

```
signature ::= [ [type {, type } ] [return type] ]
```

Où les types sont ceux des arguments et de l'éventuelle valeur retournée, dans le cas d'une fonction. Dans la définition syntaxique précédente, qui peut servir pour décorer un sous-programme avec un attribut, ou pour créer un alias, les crochets externes font partie du langage, pas de la BNF.

Le même nom peut donc désigner plusieurs sous-programmes différents. Un corollaire de cette multiplicité de sens est que les types des arguments doivent être connus lors de l'appel. Il arrive, rarement, qu'une ambiguïté apparaisse, avec des constantes littérales, le type doit alors être précisé ('0' est il un bit ou un character?).

Des exemples de procédures et fonctions surchargées peuvent être trouvés dans les bibliothèques de tous les compilateurs. Le paquetage `textio` définit, par exemple, seize versions de la procédure `READ` qui extraient un objet (nombres de tous types, bit, etc.) d'une chaîne de caractères. Une procédure de même nom est prédéfinie dans le langage pour la lecture dans un fichier. Le même nom `READ` remplace donc tous les `scanf()`, `sscanf()` du C ... et tous leurs formats possibles.

### ➤ Surcharge d'opérateurs

La surcharge d'un opérateur passe par la définition d'une fonction associée qui porte le nom de l'opérateur entre guillemets et reçoit en argument les opérandes de gauche et de droite dans cet ordre : `a + b` et `"+"(a,b)` sont des expressions équivalentes. Surcharger un opérateur rajoute un sens au symbole, cela ne modifie pas les propriétés syntaxiques attachées au symbole, la position dans l'échelle de priorité, par exemple.

Les paquetages de la bibliothèque IEEE étendent les opérateurs logiques et arithmétiques aux types multivalués et aux vecteurs de ces types interprétés comme des nombres entiers. Nous reviendrons plus en détail sur le contenu de cette bibliothèque.

L'exemple qui suit est plus un amusement que vraiment utile, il consiste à redéfinir l'addition des entiers pour que l'opérateur '+' agisse sur des chiffres en base 10 (une façon compliquée d'éviter d'écrire `mod 10`).

```
type chiffre is range 0 to 9;
function sum10 (a,b : integer) return integer ;
function "+"(a,b : chiffre) return chiffre is
```

1. Le C n'autorise pas la surcharge, il est donc impossible de définir des opérateurs arithmétiques agissant sur les nombres complexes. C++ ou VHDL permettent ce genre de choses.



```

begin
    return chiffre(sum10(integer(a),integer(b))) ;
end "+" ;

function sum10 (a,b : integer) return integer is
    variable tmp : integer range 0 to 18 ;
begin
    tmp := a + b ;
    if tmp > 9 then
        tmp := tmp - 10 ;
    end if ;
    return tmp ;
end sum10 ;

```

L'utilisation d'une fonction intermédiaire `sum10` évite une récursion infinie dans la définition de la fonction `"+"` : dans le calcul de l'expression `a + b` l'opérateur d'addition représente l'addition sur des entiers, pas sur des chiffres en base 10.

## 2.5.2 Librairies et paquetages

Une application écrite en VHDL est subdivisée en plusieurs fichiers. Chaque fichier contient une ou plusieurs unités de conception qui sont analysées dans leur ordre d'écriture par le compilateur et rangées dans des librairies sous la forme d'unités de librairies. Le mode de conservation physique des librairies n'est pas spécifié dans le langage, elle varie donc d'un système à l'autre<sup>1</sup>, chaque librairie est désignée par un nom symbolique, indépendamment de son support physique.

La règle, peu répandue dans le monde du logiciel, est que toutes les librairies liées à un système de développement sont fournies sous forme de programmes sources, même si, pour des raisons évidentes d'efficacité, les outils concernés utilisent une version précompilée de leurs librairies. Cette règle facilite grandement la portabilité des applications d'un système à l'autre : il suffit de compiler les librairies de l'ancien système sur le nouveau. Excellente en simulation, cette portabilité est moins systématique entre les outils de synthèse ; derrière les types et les attributs « maison » se cachent généralement des politiques de placement-routage qui n'ont pas de sens pour un autre compilateur<sup>2</sup>. La librairie IEEE a pour objectif d'harmoniser autant que faire se peut une relative cacophonie des premiers temps.

1. Classiquement une librairie correspond à une base de données. Le système de développement tient à jour des tables de correspondances entre les noms symboliques des librairies et leur localisation dans le système de fichiers de l'ordinateur hôte.
2. Un exemple simple : les types énumérés doivent évidemment être traduits par des codes binaires. Les attributs qui permettent de passer au synthétiseur des directives de codage (binaire naturel, gray, *one hot*, ...) varient grandement d'un système à l'autre. Avant les types `std_logic` la même confusion régnait au sujet des logiques non standards, trois-états et autres ou câblés.

### a) Fichiers sources et unités de conception

Il y a peu d'interactions entre la notion physique de fichier source et la notion logique d'unités de conception, la seule contrainte est qu'une unité de conception ne peut pas être partagée entre plusieurs fichiers sources.

Chaque unité de conception comporte deux parties :

- Une unité de librairie.
- Une spécification de contexte.

Une unité de librairie est un module primaire ou un module secondaire rattaché à un module primaire :

- Une déclaration d'entité (module primaire), que nous connaissons.
- Une déclaration de configuration (module primaire), que nous aborderons plus loin.
- Une déclaration de paquetage (module primaire), que nous décrivons dans les paragraphes suivants.
- Une architecture rattachée à une entité (module secondaire).
- Un corps de paquetage (module secondaire).

Un module secondaire doit appartenir à la même librairie que le module primaire auquel il se rattache.

Entités et architectures sont des images du monde physique ; les paquetages sont des ressources logicielles qui facilitent la lisibilité des programmes, en assurent les cohérences déclaratives, et sont des bibliothèques d'algorithmes généraux, indépendants de l'implantation matérielle ; les configurations sont des ensembles de liens qui permettent de préciser les associations entre les composants et les unités de conception qui en décrivent le fonctionnement.

#### ➤ Spécifications de contexte

La spécification de contexte précise l'environnement qui doit être connu du compilateur pour analyser une unité de librairie. Elle précise les noms des librairies utilisées (clauses `library`) et les déclarations qui doivent être visibles (clauses `use`). Par exemple :

```
library IEEE, VITAL ;
use IEEE.std_logic_1164.all, IEEE.numeric_std.all ;
use VITAL.VITAL_Primitives.all ;
```

annonce l'utilisation des librairies IEEE et VITAL et rend visibles toutes (mot clé `all`) les déclarations contenues dans les paquetages `std_logic_1164`, `numeric_std` et `vital_primitives`.

Les noms composés de la clause `use` (`use nom1.nom2.nom3`) agissent comme des filtres successifs, les deux premiers noms désignent une librairie et un paquetage qui y réside, le dernier peut désigner un objet du paquetage, spécifié par son nom, ou tous les objets du paquetage.

Une spécification de contexte est valable pour l'unité de conception qui suit et, s'il s'agit d'une unité primaire, pour toutes les unités secondaires qui s'y rapportent. Une clause use peut être rendue locale à un bloc (process, sous-programme, etc.) en la plaçant dans la zone déclarative du bloc concerné.

### ➤ Paquetages

Un paquetage (package) est un module purement logiciel, pris séparément, il ne lui correspond aucune structure matérielle. Il regroupe les éléments d'une boîte à outils dont l'assemblage dans un couple entité-architecture sert à construire le modèle d'un circuit.

Un paquetage est constitué de deux parties :

- La déclaration, partie visible de l'extérieur, contient principalement des déclarations de types, de constantes, de composants et de sous-programmes destinés à être utilisés par d'autres unités de conception :

```
package identificateur is
    déclarations de types, de fonctions,
    de composants, d'attributs,
    clause use, etc.
end [package] [identificateur] ;
```

- Le corps d'un paquetage, qui n'existe pas forcément, contient les corps des sous-programmes déclarés dans la partie visible, et des déclarations locales qui n'ont pas vocation à être visibles de l'extérieur.

```
package body identificateur is
    corps des sous-programmes déclarés.
end [package body] [identificateur] ;
```

Les programmes ci-dessous reprennent l'exemple du contrôleur de parité en déplaçant dans un paquetage fonction et procédure.

Le paquetage :

```
-- par_pkg.vhd          premier fichier source

package par_pkg is
    procedure pariproc (signal e : in bit_vector ;
        signal impair : out bit ) ;
    function parifonc (signal e : in bit_vector ) return bit ;
end package par_pkg ;

package body par_pkg is

    procedure pariproc (signal e : in bit_vector ;
        signal impair : out bit ) is
```

```

variable parite : bit := '0' ;
begin
  FOR i in e'range LOOP
    if e(i) = '1' then
      parite := not parite;
    end if;
  END LOOP;
  impair <= parite;
end procedure pariproc ;

function parifonc (signal e : in bit_vector ) return bit is
  variable parite : bit := '0' ;
begin
  FOR i in e'range LOOP
    if e(i) = '1' then
      parite := not parite;
    end if;
  END LOOP;
  return parite;
end function parifonc ;

end package body par_pkg ;

```

Le circuit :

```

-- parite.vhd      deuxième fichier source

use work.par_pkg.all ;

ENTITY ouex IS
  generic(taille : integer := 8) ;
  PORT ( a : IN BIT_VECTOR(0 TO taille - 1) ;
        s : OUT BIT );
END ouex;

ARCHITECTURE proc of ouex is
BEGIN
  pariproc(a,s) ;
END proc ;

ARCHITECTURE fonct of ouex is
BEGIN
  s <= parifonc(a) ;
END fonct ;

```

Toute unité de conception connaît, sans avoir à le spécifier, deux bibliothèques prédéfinies : `work` et `std`.

### *b) La librairie work*

La librairie *work* est, comme son nom l'indique, la librairie de travail dans laquelle sont rangées, sauf spécification explicite contraire, les unités compilées. Cette librairie est toujours ouverte et son contenu visible.

Sur certains systèmes cette librairie est quelque peu virtuelle, elle n'existe pas physiquement dans le système de fichiers, mais recouvre des liens vers des répertoires prédéfinis : vers le répertoire de travail fixé pour la session, vers le fichier en cours de compilation. Le concepteur n'a pas à se préoccuper de ces aspects qui sont définis à l'installation des logiciels.

Sur d'autres systèmes l'utilisateur est invité à créer explicitement cette librairie, lors de la première session de travail sur un projet.

Chaque politique a ses avantages et inconvénients. La première rend les choses transparentes à un premier niveau, mais un peu frustrantes au fond, l'utilisateur est peu informé des objets générés par son travail. La seconde nécessite quelques opérations d'administration système, mais a l'avantage d'une plus grande lisibilité.

### *c) La librairie std*

Tout système de développement en VHDL est assorti d'une librairie *std*. Cette librairie contient deux paquetages : *standard* et *textio*. Avant de les décrire mentionnons une précaution importante. Les sources de ces deux paquetages sont fournis à titre de documentation et de conformité à la norme. Mais la majorité des compilateurs incluent certaines des fonctions définies dans ces paquetages dans le noyau du compilateur. Il est formellement déconseillé de modifier quoique ce soit dans le contenu de la librairie *std*, sous peine de s'exposer à des incohérences graves de fonctionnement des logiciels.

#### » Le paquetage standard

Le paquetage *standard* est toujours visible, il n'est donc pas nécessaire de prévoir une clause *use* pour l'utiliser.

Il contient les définitions des types fondamentaux du langage : *boolean*, *bit*, *character*, *integer*, *real*, *time*, *natural* (*integer range 0 to integer'high*), *positive* (*integer range 1 to integer'high*), *string*, *bit\_vector* et quelques autres qui concernent les traitements d'erreurs et les manipulations de fichiers.

La consultation de ce paquetage renseigne sur certains détails liés à une implémentation, comme la précision des nombres réels ou la définition des caractères non *ascii* comme les lettres accentuées.

#### » Le paquetage textio

Le paquetage *textio* définit des types et des procédures qui facilitent les manipulations de fichiers en mode texte. Son utilisation nécessite la clause *use std.textio.all*.

Il définit également un fichier d'entrée standard (*input*) et un fichier de sortie standard (*output*), tous deux en mode texte. Par défaut ces deux fichiers correspondent

à la console de la session de travail et permettent l'affichage de messages et la saisie de commandes.

Le principe général adopté pour accéder à un fichier est de procéder en trois temps ; par exemple en lecture :

- Ouverture du fichier en mode `read_mode`.
- Lecture, ligne par ligne, du fichier vers un tampon alloué dynamiquement en mémoire. L'accès au tampon se fait au moyen d'un pointeur dont le type (`line`) est défini par le paquetage. La procédure de lecture d'une ligne est déclarée par :

```
procedure readline(file f : text; L : out line) ;
```

- Transfert des données du tampon vers leur destination finale au moyen des procédures `read()`. Le mécanisme de surcharge définit une procédure `read()` pour les principaux types de base du langage. Ces procédures fournissent un compte rendu de la bonne ou mauvaise issue de l'opération, ce qui permet de détecter des erreurs de format éventuelles. La procédure de lecture d'un entier, à titre d'illustration, est déclarée :

```
procedure read(L : inout line; VALUE : out integer;
GOOD : out boolean) ;
```

Très simples de mise en œuvre, les opérations sur les fichiers sont limitées au strict nécessaire : l'accès séquentiel, qui permet, entre autres, d'initialiser une mémoire, de lire une liste de stimuli, d'enregistrer une séquence de résultats. Évidemment non synthétisables, les opérations sur les fichiers sont cependant acceptées par certains outils de synthèse comme moyen d'initialisation de constantes (mémoires ROM).

Nous aurons l'occasion de présenter des exemples d'utilisation des fichiers dans un paragraphe ultérieur (2-7).

#### d) La librairie IEEE et la portabilité des sources

Un compilateur VHDL est toujours assorti d'une librairie, décrite par des paquets, qui offre à l'utilisateur des outils variés :

- Définitions de types, et fonctions de conversions entre types : VHDL est un langage objet, fortement typé. Aucune conversion de type implicite n'est autorisée dans les expressions, mais une librairie peut offrir des fonctions de conversion explicites, et redéfinir les opérateurs élémentaires pour qu'ils acceptent des opérandes de types variés. Un bus, par exemple, peut être vu dans le langage, comme un vecteur (tableau à une dimension) de bits, et il est possible d'étendre les opérateurs arithmétiques et logiques élémentaires pour qu'ils agissent sur un bus, vu comme la représentation binaire d'un nombre entier.
- Une porte trois états, par exemple, sera vue, dans une architecture, comme un composant dont l'un des ports véhicule des signaux de type particulier : aux deux états logiques vient se rajouter un état haute impédance. L'emploi d'un tel opérateur dans un schéma nécessite, outre la description du composant, une fonction de conversion entre signaux binaires et signaux trois états.

- Un simulateur doit pouvoir résoudre, ou indiquer, les conflits éventuels. Les signaux utilisés en simulation ne sont pas, pour cette raison, de type binaire : on leur attache un type énuméré plus riche qui rajoute aux simples valeurs '0' et '1' la valeur inconnue 'X', des nuances de force entre les sorties standards et les sorties collecteur ouvert, etc.

Les types définis dans le langage sont peu nombreux et trop sommaires pour rendre compte des cas réels, même courants. Mais VHDL permet de créer à sa guise des types et les opérations afférentes. On a vu ainsi fleurir les bibliothèques maison, chaque système de développement offrait ses propres types pour prendre en compte les conflits, les opérateurs câblés, les connexions en bus ; tous différents et incompatibles entre eux, cela va sans dire.

La bibliothèque IEEE joue un rôle fédérateur et remplace tous les dialectes locaux. En cours de généralisation, y compris en synthèse, elle définit un type de base à neuf états, `std_ulogic` (présenté précédemment comme exemple de type énuméré) et des sous-types dérivés simples et structurés (vecteurs). Des fonctions et opérateurs surchargés permettent d'effectuer des conversions et de manipuler les vecteurs comme des nombres entiers.

À l'heure actuelle la bibliothèque IEEE comporte trois paquetages dont nous examinons plus en détail certains aspects au paragraphe 2-7 :

- `std_logic_1164` définit les types, les fonctions de conversion, les opérateurs logiques et les fonctions de recherches de fronts `rising_edge()` et `falling_edge()`.
- `numeric_bit` définit les opérateurs arithmétiques agissant sur des `bit_vector` interprétés comme des nombres entiers.
- `numeric_std` définit les opérateurs arithmétiques agissant sur des `std_logic_vector` interprétés comme des nombres entiers.

Nous ne pouvons que conseiller vivement l'usage de cette bibliothèque en lieu et place de son équivalent lié à un fournisseur particulier de logiciel.

Le programme ci-dessous illustre l'utilisation de la bibliothèque IEEE dans la modélisation d'un compteur binaire synchrone, remise à zéro comprise, à sorties trois états. On notera la définition du type `unsigned`, qui permet de représenter un nombre entier naturel par un vecteur d'éléments binaires. L'indice de ce vecteur doit être descendant (`downto`), reproduisant par là l'écriture polynomiale usuelle des nombres où l'on note de gauche à droite les chiffres dans un ordre de poids décroissants.

```
library ieee ;
use ieee.std_logic_1164.all, ieee.numeric_std.all ;
-- numeric_std contient la définition du type unsigned :
-- type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
-- Les opérateurs arithmétiques usuels sont surchargés,
-- par exemple :
-- function "+" (L, R: UNSIGNED) return UNSIGNED;
-- function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
-- function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
```

```

entity compteur_tri is
    generic(taille : integer := 10) ;
    port( hor,raz,en,oe : in std_ulogic ;
          compte : out unsigned(taille-1 downto 0) ) ;
end compteur_tri ;

architecture ieee_lib of compteur_tri is
    signal etat : unsigned(taille-1 downto 0) ;
begin
    compte <= etat when oe = '1' else (others => 'Z') ;
    compteur : process
    begin
        wait until rising_edge(hor) ;
        if raz = '1' then
            etat <= (others => '0') ;
        elsif en = '1' then
            etat <= etat + 1 ;
        end if ;
    end process compteur ;
end ieee_lib ;

```

Le modèle ci-dessus est accepté tant en simulation qu'en synthèse par tout compilateur qui inclut le standard IEEE 1076.3.

## 2.6 CONSTRUCTION HIÉRARCHIQUE

La vision d'un circuit complexe comme l'assemblage d'opérateurs plus simples, vision qui n'est d'ailleurs propre ni à VHDL ni à l'électronique, correspond à une démarche descendante bien établie (*top down design*). Au sommet de la hiérarchie on trouve une perception globale, dans laquelle n'apparaissent pas les détails de fonctionnement, seuls sont visibles des blocs fonctionnels et leurs canaux de communication. Chaque bloc est lui-même le sommet d'une hiérarchie locale. La subdivision s'arrête quand la dimension des modules élémentaires est suffisamment simple pour pouvoir être pensée comme un tout, ou quand ces modules sont les composants ultimes de la technologie employée.

Plus encore qu'avec le découpage d'un algorithme en sous-programmes, la vision est ici spatiale. Ce n'est pas un hasard si de nombreux outils classiques de CAO passent par une interface graphique pour définir les éléments de la hiérarchie. De nombreux logiciels permettent d'ailleurs de passer d'une représentation graphique à une représentation textuelle et vice versa. La limitation principale des représentations graphiques est leur caractère figé : pour qu'un schéma soit une illustration claire de la réalité, à chaque bloc fonctionnel doit correspondre un objet dessiné. Cette contrainte peut parfois être assouplie, des « macros » graphiques sont pensables, mais la lisibilité du schéma en souffre fortement.



VHDL autorise la création d'assemblages paramétrables, des boucles et des tests permettent de construire un schéma par une sorte d'algorithme spatial. Les correspondances entre les composants d'un niveau hiérarchique et les unités de conception, qui en décrivent le fonctionnement aux niveaux inférieurs, sont modifiables dans des configurations, sans que les objets concernés aient eux-mêmes à être altérés. *Generate* et *configuration* sont les mots clés de cette partie. Nous les aborderons après avoir complété quelques propriétés des blocs.

### 2.6.1 Blocs externes et blocs internes

Un composant instancié fait référence à un bloc externe : un autre couple entité architecture, qui ne communique que par ses ports d'accès.

Un processus ou un bloc explicite (créé par l'instruction `block`) est interne : il est en communication directe avec les signaux de l'architecture environnante.

#### a) *Localité des noms*

Un bloc externe, instancié comme composant, ne connaît pas même l'existence de l'architecture environnante ; les noms utilisés dans les deux parties sont donc complètement indépendants les uns des autres. Le bloc externe peut être instancié dans d'autres architectures, testé indépendamment, il est une unité de conception complètement autonome.

Un bloc interne peut déclarer des noms qui lui sont propres, qui sont invisibles de l'extérieur, mais il hérite des noms déclarés dans son environnement. En cas de conflit (homonymie) les noms locaux masquent les noms extérieurs. Un bloc interne est attaché à son environnement, plus simple par nature, il est moins souple qu'un bloc externe, peu ou pas réutilisable ailleurs.

#### b) *Paramètres génériques*

Les paramètres génériques sont le premier outil de l'algorithmique spatiale évoquée précédemment. Ils permettent de différer, ou modifier, certains choix de dimension comme la largeur d'un bus, le nombre de cellules d'un registre, le nombre de chiffres d'un compteur, etc.

Il est clair qu'à l'assemblage final du projet, tous les paramètres doivent avoir des valeurs connues, ce sont des constantes, pas des objets dont le contenu peut être modifié dynamiquement par le simulateur ou le circuit synthétisé.

Pour réaliser une décade d'un compteur décimal, il est possible, par exemple, d'utiliser un compteur générique :

```
entity mod_cpt is
    generic(modulo : integer := 16) ;
    port(clk, clear, cen : in bit ;
          cout : out integer range 0 to modulo-1 ;
          rco : out bit ) ;
end mod_cpt ;
```

```

architecture comporte of mod_cpt is
    signal state : integer range 0 to modulo-1 ;
begin
    rco <= cen when state = modulo-1 else '0' ;
    cout <= state ;
    cpt : process
    begin
        wait until clk = '1' ;
        if clear = '1' then
            state <= 0 ;
        elsif cen = '1' then
            state <= (state + 1) mod modulo ;
        end if ;
    end process cpt ;
end comporte ;

```

Ce compteur étant compilé dans une librairie genlib, il peut être instancié comme compteur modulo 10 pour constituer une décade :

```

USE work.decimal.all;

ENTITY decade IS
    PORT (clk, en, raz : in bit ;
          count: OUT chiffre ;
          dix : OUT bit) ;
END decade ;

library genlib ;

architecture instance of decade is
begin
    dec : mod_cpt generic map (modulo => 10)
        port map (clk => clk, clear => raz, cen => en,
                  cout => integer(count), rco => dix);
end instance ;

```

Où les types chiffre et nombre ainsi que le composant mod\_cpt sont définis par :

```

package decimal is
    type chiffre is range 0 to 9 ;
    type nombre is array (integer range <>) of chiffre ;
    -- ...
    component mod_cpt is
        generic(modulo : integer := 16) ;
        port(clk, clear, cen : in bit ;
              cout : out integer range 0 to modulo-1 ;
              rco : out bit ) ;
    end component mod_cpt ;
end decimal ;

```

Lors de l'instanciation du composant `mod_cpt` il est nécessaire de prévoir une conversion entre les types `chiffre` et `integer` (`cout => integer(count)`), le langage n'acceptant pas de conversions implicites, même proches.

### 2.6.2 L'instruction `generate`

Nous avons vu (§ 2-3.1), comme exemple d'instanciation de composant, la description d'un compteur décimal à deux chiffres. Alors qu'il est simple, par un simple générique, de créer un compteur binaire de dimension paramétrable, le cas du compteur décimal est un peu moins trivial : il faut respecter le découpage en tranches de quatre chiffres binaires attachés à chaque chiffre décimal. Une première solution consisterait à faire appel à un algorithme du monde séquentiel au moyen d'une double boucle, dans un processus ou un sous-programme.

La solution présentée ici fait appel à l'instruction `generate`, qui permet de créer directement des boucles et des tests spatiaux dans le monde concurrent, donc, entre autres, de créer un schéma dont la structure obéit à certaines règles.

#### a) *Boucles et tests dans l'univers concurrent*

L'instruction concurrente `generate` définit un nombre arbitraire de blocs implicites :

```
generate_statement ::=
    etiquette : generation_scheme generate
        [ zone_declarative_de_bloc begin ]
        instructions_concurrentes
    end generate [etiquette] ;

generation_scheme ::=
    for parametre in domaine
    if condition
```

Si aucune déclaration locale n'est nécessaire au bloc d'instructions contrôlées, il est inutile de marquer le début de bloc par `begin`. L'étiquette, qui définit le nom du bloc, est obligatoire.

Le compteur décimal peut être construit en dupliquant le code du processus associé à une décade, autant de fois que nécessaire, en assurant un chaînage des retenues des poids faibles vers les poids forts (une décade s'incrémente si toutes celles de poids inférieurs sont à 9) :

```
-- compteur décimal à mise à zéro synchrone
-- nombre de chiffres paramétrables

use work.decimal.all; -- définit chiffre et nombre

entity cntdec is
    generic ( taille : integer := 2 ) ;
    -- permet de fixer le nombre de décades, 2 par défaut
    port ( clk, en, zero : in bit ;
          sortie : out nombre (taille-1 downto 0) ;
          carry : out bit) ;
```

```

end cntdec ;

architecture multibloc of cntdec is
    signal retenue : bit_vector (taille downto 0) ;
begin
    retenue(0) <= en ;
    carry <= retenue(taille) ;
    ci : for i in 0 to taille - 1 generate
        signal countTemp : chiffre ;
    begin
        sortie(i) <= countTemp ;
        retenue(i+1) <= retenue(i) when countTemp = 9 else '0' ;
        incre : process
        begin
            wait until clk = '1' ;
            if zero = '1' then
                countTemp <= 0 ;
            elsif retenue(i) = '1' then
                countTemp <= (countTemp + 1) mod 10 ;
            end if ;
        end process incre ;
    end generate ci ;
end multibloc ;

```

Nous avons mis en gras les frontières du bloc `generate` pour faciliter son identification.

Le signal `countTemp`, qui est déclaré localement, est physiquement dupliqué autant de fois que nécessaire (`taille`), il correspond à un registre synchrone de quatre bits qui contient la valeur d'un chiffre décimal. Le code du processus, dupliqué lui aussi, engendre dans le circuit les commandes de ces registres. La boucle `for` génère un bloc pour chaque valeur de son indice ; chacun de ces blocs implicites porte un nom construit à partir du label de l'instruction `generate` et de l'indice de boucle : `ci(i)`. Ce nom peut être utile lors de la construction d'une configuration.

### b) Instanciations multiples : des schémas algorithmiques

Dans l'exemple précédent nous avons volontairement mis dans le corps de l'instruction `generate` plusieurs instructions concurrentes, dont un processus explicite, pour illustrer la généralité de la construction.

Dans de nombreux cas `generate` sert à contrôler l'instanciation de composants, dans une construction hiérarchique. Si la description d'une décade est mise dans une unité de conception autonome, le code du compteur décimal s'en trouve évidemment simplifié<sup>1</sup> :

```

architecture struct of cntdec is
    signal retenue : bit_vector (taille-1 downto 0) ;
begin

```

1. Le composant `decade` doit être déclaré, par exemple dans le paquetage `decimal`.

```

carry <= retenue(taille-1) ;
ci : for i in 0 to taille - 1 generate
  c0 : if i = 0 generate
    ci0 : decade port map (clk,en,zero,
                           sortie(i),retenue(i));
  end generate c0 ;
  cn : if i > 0 generate
    cin : decade port map (clk,retenue(i-1),zero,
                           sortie(i),retenue(i)) ;
  end generate cn ;
end generate ci ;
end struct ;

```

Dans cet exemple nous avons traité à part l'étage de poids faible, pour lequel l'autorisation de comptage ne provient pas des retenues, ce qui nous donne l'occasion d'utiliser les deux structures de contrôle de l'instruction `generate`.

Paramètres génériques et instruction `generate` sont deux outils distincts, souvent employés de concert, il est vrai. Certains auteurs les réunissent dans un concept global de généricité.

Si les programmes précédents semblent quelque peu lourds, à première lecture, il est utile de se souvenir qu'ils permettent, par exemple, de créer une échelle de comptage sur 15 chiffres sans aucune modification du code. Leur équivalent graphique ne sera, dans ce cas, guère plus léger.

### 2.6.3 Configuration d'un projet

Jusqu'ici les liens entre les composants d'un niveau hiérarchique et les unités de conception qui leur sont rattachées étaient implicites. Nous ne précisons rien, ce qui revient à supposer que dans la librairie de travail existe une entité qui présente la même interface que le composant : nom identique, mêmes paramètres génériques s'il y en a et mêmes ports d'accès.

Pratique dans les projets simples, ce lien implicite devient une limitation dans des applications plus complexes. Dans ce cas il est souhaitable que les composants et les unités de conception bénéficient d'une indépendance de noms. Cette indépendance est rendue possible par l'utilisation de directives de configuration, sortes de tables de correspondances explicites.

Deux niveaux de configurations peuvent être employés, qui ne doivent pas entrer en conflit sous peine d'erreur :

- Les spécifications de configuration d'un composant sont locales au bloc où est instancié le composant.
- La déclaration de configuration d'une entité contient des directives générales, hiérarchisées, qui permettent de préciser quelle architecture doit être employée, et, récursivement, pour tous les blocs et composants de cette architecture, quelles sont leurs configurations.

Nous examinerons successivement ces deux niveaux d'organisation d'un projet en reprenant l'exemple d'un compteur décimal. Nous admettons que les unités de

conception appelées par les configurations existent, le code correspondant n'est pas donné pour des raisons de concision.

### a) Spécification de la configuration d'un composant

En l'absence d'autre précision, le nom d'un composant renvoie à une entité de même nom dans la librairie `work`, dont les paramètres génériques et les ports correspondent à ceux déclarés pour le composant et dont le compilateur prend la dernière architecture analysée.

Une spécification de configuration permet de modifier ces associations par défaut en précisant quelle unité de conception doit être employée pour un composant, et, si nécessaire, de fixer les valeurs des paramètres génériques éventuels ainsi que la correspondance entre les ports du composant et ceux de l'entité associée :

```
for liste_labels : composant [use précision_entité]
    [generic map ( ... )]
    [port map ( ... )] ;

précision_entité ::=
    entity librairie.entité [(architecture)]
    | configuration librairie.configuration
```

Le terme de configuration, s'il est utilisé, fait référence à une unité primaire de conception (voir ci-dessous). Les instructions `port map` et `generic map` utilisent les listes d'association entre paramètres formels (ceux de l'entité) et paramètres réels (ceux du composant), avec la même syntaxe que celle qui associe les ports du composant aux équipotentielles de l'architecture environnante. La liste des labels contient la ou les étiquettes d'instanciation concernées, ou les mots clés `all` ou `others`.

Par exemple :

```
ci : for i in 0 to taille - 1 generate
    c0 : if i = 0 generate
        for ci0 : decade use entity work.decade(vecteur) ;
        ci0 : -- instanciation : même code que précédemment
    end generate c0 ;
    cn : if i > 0 generate
        for cin : decade use entity work.decade(surcharge) ;
        cin : -- instanciation : même code que précédemment
    end generate cn ;
end generate ci ;
```

Permet d'instancier, pour différents exemplaires du composant `decade`, deux architectures différentes de la même unité de conception `decade`.

### b) Déclaration de configuration d'une entité

Une déclaration de configuration se rapporte à une entité. Elle permet de choisir une architecture, et, pour chaque bloc de cette architecture, de préciser sa configuration, par un mécanisme de directives emboîtées, qui reproduisent la hiérarchie du schéma :



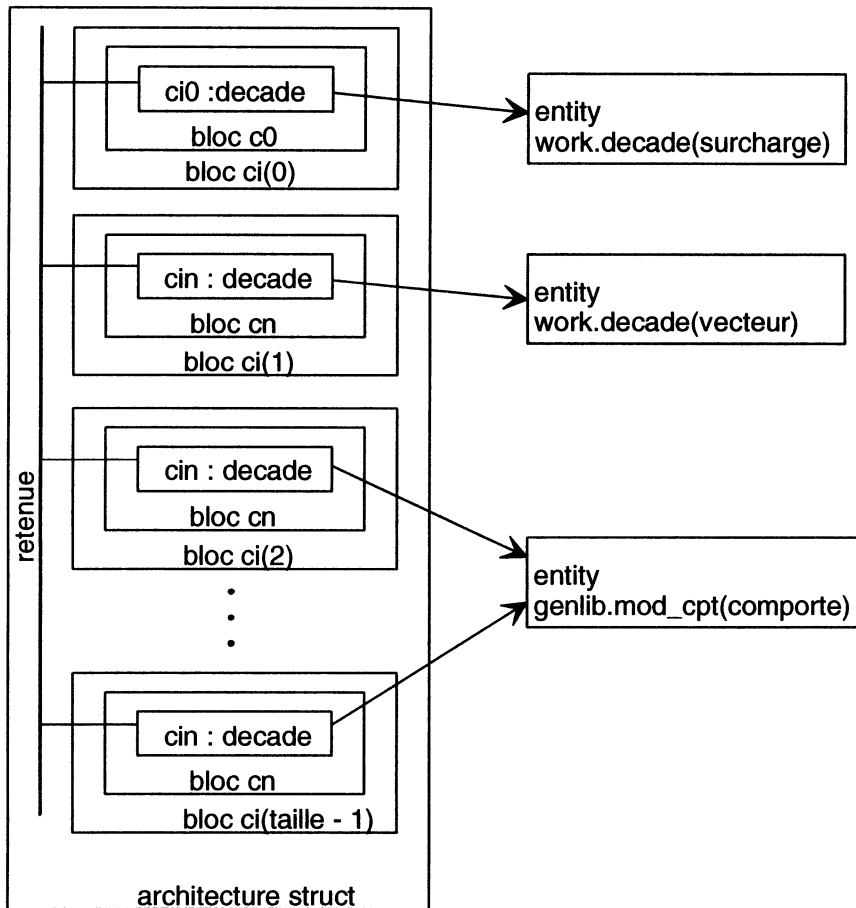


Figure 2-23 Compteur décimal, vision structurelle.

```

        end for ;
    end for ;
end for ;
for ci(2 to taille - 1)
    for cn
        for cin:decade -- noms différents
            use entity genlib.mod_cpt(comporte)
            generic map(modulo => 10)
            port map (clk => clk, cen => en,
                    clear => raz,
                    cout => integer(count),
                    rco => dix ) ;

            end for ;
        end for ;
    end for ;
end for ;

```



```
end for ;  
end configuration cntstruc ;
```

On remarquera la conversion de type, entre types voisins, dans une instruction `port map`, et la façon de repérer le nom d'un bloc implicite d'une instruction `generate`.

Une configuration peut être simulée et synthétisée. Elle représente une version d'un projet obtenue par l'assemblage d'éléments divers, sans obligation de respecter des règles de concordance de noms.

## 2.7 MODÉLISATION ET SYNTHÈSE

Langage de modélisation des circuits intégrés complexes, VHDL est devenu un outil de synthèse. Sans parler des logiciels de conception d'ASICs sur stations de travail, la plupart des fabricants de circuits programmables offrent des solutions VHDL, plus ou moins complètes, qui ne nécessitent qu'un équipement de type PC.

### 2.7.1 Tout ce qui est synthétisable doit être simulable

Tout module synthétisable est simulable, et les résultats de simulations sont identiques à ceux que l'on observera dans le vrai circuit. Telle est la règle, généralement bien respectée. Tous les logiciels connus des auteurs peuvent cependant être pris en défaut, nous savons, par des constructions « à la limite », transformer des bascules *edge* (en simulation) en bascules *latch* dans le circuit, et réciproquement<sup>1</sup>.

Ne noircissons pas, le rôle de cet ouvrage est justement de prévenir ce genre de mésaventures, avant toute réalisation. La règle d'or est dans la limpidité du code. Un programme bien construit est toujours compilé correctement.

### 2.7.2 La réciproque n'est pas vraie mais ...

Toutes les constructions du langage ne sont pas synthétisables, et il n'y a rien là que de tout à fait normal. Certaines ne le sont pas par nature, elles ne prétendent pas avoir de sens dans un circuit, d'autres ne le sont pas pour un logiciel donné, à une époque donnée, mais il ne serait pas absurde, dans le principe, qu'elles le deviennent un jour. Tenant compte de ces remarques, nous classerons ci-dessous les catégories non synthétisables en quatre familles, illustrées par des exemples dont nous ne garantirons pas l'exhaustivité.

- Constructions non synthétisables par nature : les manipulations de pointeurs, les manipulations de types non contraints (tableaux de dimensions non spécifiées<sup>2</sup>, nombres sans limitation de leur domaine de définition), la modélisation des

1. C'est un peu plus difficile.

2. Cela n'empêche pas d'utiliser des fonctions dont les arguments sont non contraints. Il suffit qu'à l'élaboration finale de l'unité de conception, donc à l'appel de la fonction, les dimensions soient connues.

retards, les tests de violations de *timing*, les envois de messages sur la console, les lectures de directives au clavier, les références au temps du simulateur (le type *time* est refusé en tant que tel par de nombreux outils de synthèse), les accès dynamiques aux fichiers.

- Constructions non synthétisables, mais acceptées par plusieurs logiciels de synthèse comme outils généraux de conception : certains accès aux fichiers. La lecture fichiers de données permet, par exemple, d'initialiser des tables ou des mémoires. Vues du code synthétisable, les données issues d'un fichier n'ont de sens que si ce sont des constantes.
- Constructions non synthétisables en raison de la complexité sous-jacente : essentiellement les opérations arithmétiques. Tous les systèmes imposent des limites aux opérateurs et aux types d'opérandes acceptés dans ce domaine. Ces limites varient grandement d'un compilateur à l'autre et progressent d'une version à l'autre d'un même compilateur. Seule une lecture attentive de la documentation spécifique de l'outil permet de les connaître. En tout état de cause rappelons que les opérations arithmétiques « brutales » génèrent rapidement des schémas extrêmement complexes, qui se heurteront éventuellement à la réalité du circuit cible.
- Constructions synthétisables qui font appel à un sens caché. L'affaire est plus délicate : à la définition de certains objets est attachée une signification qui pilote le synthétiseur, à condition qu'on ait réussi à lui faire comprendre ce que l'on souhaite obtenir. Prenons un exemple.

Pour réaliser une porte à sortie trois états il peut venir à l'esprit de créer un type et une fonction de conversion associée :

```
package troisetpkg is
  type z_0_1 is ('0','1','Z') ;
  function to_z_0_1(e, oe : bit) return z_0_1 ;
end package troisetpkg ;

package body troisetpkg is
  function to_z_0_1(e, oe : bit) return z_0_1 is
  begin
    if oe = '0' then
      if e = '0' then
        return '0' ;
      else
        return '1' ;
      end if ;
    else
      return 'Z' ;
    end if ;
  end to_z_0_1 ;
end package body troisetpkg ;

use work.troisetpkg.all ;

entity tri_buffer is
  port(e, oe : in bit ;
```

```

        s : out z_0_1 ) ;
end tri_buffer ;

architecture test of tri_buffer is
begin
    s <= to_z_0_1(e,oe) ;
end test ;

```

Ce programme est parfaitement synthétisable, mais ne produit pas vraiment le résultat escompté. Le type énuméré est codé sur deux éléments binaires, qui prennent trois des quatre valeurs possibles ("00", "01", "10" et "11") en fonction des entrées, sans l'ombre d'état haute impédance...

La définition d'un sous-type, héritier du type logique multivalué `std_ulogic`, change tout, dans le bon sens :

```

library ieee ;
use ieee.std_logic_1164.all ;

package troisetpkgieee is
    subtype z_0_1 is std_ulogic range '0' to 'Z' ;
    function to_z_0_1(e, oe : bit) return z_0_1 ;
end package troisetpkgieee ;

package body troisetpkgieee is
    function to_z_0_1(e, oe : bit) return z_0_1 is
    begin
        -- même programme source que précédemment
    end package body troisetpkgieee ;

use work.troisetpkgieee.all ;

entity tri_bufl is
    port(e, oe : in bit ;
        s : out z_0_1 ) ;
end tri_bufl ;

architecture test of tri_bufl is
begin
    s <= to_z_0_1(e,oe) ;
end test ;

```

Le circuit synthétisé réalise bien un tampon trois-états, dont la sortie pilote une équipotentielle simple. L'état haute impédance est commandé par l'entrée `oe`. En synthèse certains types ont un sens caché mais précis, le rôle de la norme IEEE 1164 est de pousser tous les concepteurs de logiciels à tenir le même langage. Les deux programmes précédents ont évidemment strictement le même comportement en simulation, le contraire n'aurait aucun charme.

Un compilateur de synthèse peut réagir diversement face à une instruction non synthétisable : la traiter comme une erreur, l'ignorer si le programme a malgré tout un sens, voire ne pas détecter le piège, ce qui est assurément dangereux. Les restrictions

apportées par un logiciel à la norme du langage sont bien évidemment documentées, souvent sous forme de BNF.

#### a) Penser « circuit »

Les poubelles de nos disques durs sont remplies de programmes dont les algorithmes seraient justes, s'ils étaient traduits en C et exécutés de point d'arrêt en point d'arrêt (ou pire, en pas à pas) sur un ordinateur. Mais les résultats fugaces de ces algorithmes disparaissent en un temps nul, ne sortent pas de la boîte qui les a fait naître. En cas de doute, le recours est de prendre papier et crayon et de dessiner des rectangles avec des flèches qui rentrent, d'autres qui sortent, en bref ébaucher un schéma.

Diviser un projet en (petits) blocs autonomes, nommer les signaux qui permettent à ces blocs de dialoguer, imaginer l'architecture physique du circuit en cours d'élaboration représente une bonne partie du travail initial de conception, VHDL ou pas.

#### b) De l'ordre dans les horloges

Tous les outils de synthèse attendent une identification claire des signaux d'horloges. Les instructions de test des fronts doivent être séparées de celles qui surveillent les commandes. Par exemple :

```
entity T_edge is
  port ( T,hor : in bit;
        s : out bit);
end T_edge;

architecture mauvaise of T_edge is
  signal etat : bit ;
begin
  s <= etat ;
  process (hor)
  begin
    if hor'event and hor = '1' and T = '1' then
      etat <= not etat ;
    end if ;
  end process ;
end mauvaise ;
```

est une mauvaise bascule : la même instruction recherche les fronts montants d'horloge et teste la valeur de l'entrée T.

La version correcte du programme sépare ces deux actions, ce qui correspond d'ailleurs à la structure physique du circuit :

```
architecture bonne of T_edge is
  signal etat : bit ;
begin
  s <= etat ;
  process (hor)
  begin
    if hor'event and hor = '1' then -- recherche du front
      if T = '1' then
```

```

        etat <= not etat ; -- action synchrone
    end if ;
end if ;
end process ;
end bonne ;

```

La même remarque peut être faite avec l'instruction `wait`, si on l'utilise, elle ne doit exprimer que l'attente du front actif de l'horloge, à l'exclusion de tout autre test. Rappelons ici que certains compilateurs de synthèse ne sont pas très regardants en ce qui concerne les listes de sensibilités des processus. C'est évidemment une mauvaise habitude que d'en profiter pour faire de même.

### 2.7.3 Pilotes multiples et fonctions de résolution

Les sorties standards des circuits numériques se comportent, en première approximation, comme des sources de tension. Ces sorties ne peuvent en aucun cas être connectées en parallèle. Dans le langage VHDL cette règle est traduite par l'unicité du pilote d'un signal ordinaire : la valeur véhiculée par un signal est fixée par un processus unique auquel est associé un *driver*, également unique. Tout non-respect de cette règle constitue une faute de syntaxe.

Les connexions en bus, ou la réalisation de fonctions logiques câblées utilisent des opérateurs dont la structure particulière des étages de sortie permet la mise en parallèle de ces dernières. Qui dit mise en parallèle dit conflit possible, d'où nécessité, dans le cadre de la modélisation, de résolution du conflit.

#### a) Conflits

Le principe de la résolution des conflits passe par la création d'une fonction dite de résolution. Les signaux qui sont destinés à accepter des pilotes multiples portent le nom de signaux résolus. Le mécanisme est illustré par la figure 2-24 : à chaque processus (explicite ou implicite) qui contient une affectation d'un signal résolu est associé un pilote de ce signal.

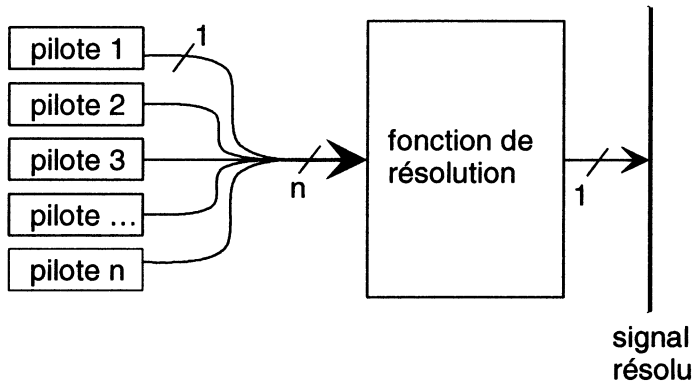
Quand un pilote effectue une transaction, une demande de modification de la valeur du signal, le noyau du simulateur appelle la fonction de résolution dont le seul argument d'entrée est le vecteur obtenu par la concaténation des valeurs de toutes les sources connectées au signal résolu. La fonction retourne une valeur unique, qui est celle que doit effectivement prendre le signal.

L'exemple qui suit explicite en détail la démarche de création de signaux résolus : on commence par créer un type énuméré (`x_0_1`), qui peut prendre les trois valeurs 'x', '0' et '1', et un type vecteur associé. Une fonction de conversion (`to_x_0_1`) permet de passer du type `bit` au type `x_0_1` pour faciliter les tests, et une fonction de résolution (`collision`) retourne le résultat de l'analyse du vecteur qui lui est passé en argument :

```

package resolution is
    type x_0_1 is ('x','0','1') ;
    type x_0_1_vector is array(natural range <>) of x_0_1 ;
    function to_x_0_1(e : bit) return x_0_1 ;

```



**Figure 2-24** Pilotes multiples et fonction de résolution.

```

function collision(e : x_0_1_vector) return x_0_1 ;
end package resolution ;

package body resolution is
  function to_x_0_1(e : bit) return x_0_1 is
  begin
    if e = '0' then return '0' ;
    else return '1' ;
    end if ;
  end to_x_0_1 ;

  function collision(e : x_0_1_vector) return x_0_1 is
    variable valeur : x_0_1 := 'x' ;
  begin
    if e'length > 0 then
      valeur := e(e'left) ; -- valeur du premier pilote
      for i in e'range loop
        if valeur /= e(i) then
          valeur := 'x' ;
        end if ;
      end loop ;
    end if ;
    return valeur ;
  end collision ;
end package body resolution ;

```

Pour créer un signal résolu, il suffit de mentionner le nom de la fonction de résolution dans la déclaration d'un signal :

```

use work.resolution.all ;

entity conflit is end conflit ;

architecture exemple of conflit is

```

```

    signal a,b : bit := '0' ;
    signal s : collision x_0_1 ; -- signal résolu
begin
    s <= to_x_0_1(a) ;
    s <= to_x_0_1(b) ;
    a <= '1' after 10 ns, '0' after 20 ns, '1' after 30 ns ;
    b <= '1' after 20 ns ;
end exemple ;

```

La variable valeur de la fonction `collision` est initialisée par défaut au contenu que doit retourner la fonction de résolution quand tous les pilotes sont déconnectés. Ce type de situation peut se rencontrer, notamment, si les signaux résolus sont contrôlés par une expression de garde d'un bloc.

Une méthode plus élégante que la précédente consiste à créer un sous-type résolu, utilisé dans les déclarations des signaux correspondants :

```

package resolution is
    type x_0_1 is ('x','0','1') ;
    type x_0_1_vector is array(natural range <>) of x_0_1 ;
    function to_x_0_1(e : bit) return x_0_1 ;
    function collision(e : x_0_1_vector) return x_0_1 ;
    subtype res_x_0_1 is collision x_0_1 ;
end package resolution ;

-- ...

architecture exemple of conflit is
    signal a,b : bit := '0' ;
    signal s : res_x_0_1 ;
begin
    -- ...
end exemple ;

```

La figure 2-25 fournit le résultat obtenu en simulation.

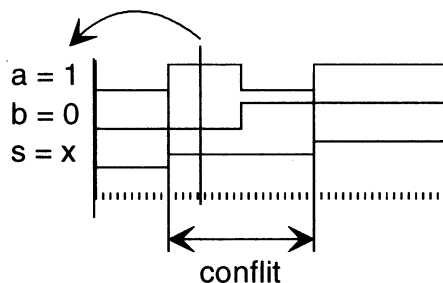


Figure 2-25 Exemple de conflit.

**b) Sorties non standard (trois états et collecteurs ouverts)**

L'utilisation des signaux résolus dans la modélisation des opérateurs à sorties trois états passe par une analyse un peu plus élaborée des conflits. On rajoute un état haute impédance, 'z', aux valeurs possibles d'une équipotentielle. Quand deux pilotes projettent des valeurs différentes pour un signal, le résultat est inconnu si les deux valeurs correspondent à des sorties en basse impédance ; si les valeurs projetées sont de forces différentes, la plus forte, c'est-à-dire celle qui correspond à l'impédance la plus faible, l'emporte.

Une méthode simple consiste à consigner tous les cas de conflits possibles dans une table. La fonction de résolution obtient le résultat d'un conflit par consultation de cette table :

```

package troisetats is
  type x_0_1_z is ('x','0','1','z') ;
  type x_0_1_z_vector is array(natural range <>) of x_0_1_z ;
  function to_x_0_1_z(e : bit) return x_0_1_z ;
  function resoudre(e : x_0_1_z_vector) return x_0_1_z ;
  subtype res_x_0_1_z is resoudre x_0_1_z ;
end package troisetats ;

package body troisetats is
  function to_x_0_1_z(e : bit) return x_0_1_z is
  begin
    if e = '0' then return '0' ;
    else return '1' ;
    end if ;
  end to_x_0_1_z ;

  type x_0_1_z_table is array(x_0_1_z,x_0_1_z) of x_0_1_z;
  constant resolution_table : x_0_1_z_table := (
    --
    -- | x   0   1   z       | |
    -- -----
    ( 'x', 'x', 'x', 'x' ), -- | x |
    ( 'x', '0', 'x', '0' ), -- | 0 |
    ( 'x', 'x', '1', '1' ), -- | 1 |
    ( 'x', '0', '1', 'z' ) -- | z |
    );

  function resoudre(e : x_0_1_z_vector) return x_0_1_z is
    variable valeur : x_0_1_z := 'z' ; -- par défaut
  begin
    if e'length = 1 then
      valeur := e(e'left) ;
    else
      for i in e'range loop
        valeur := resolution_table(valeur, e(i));
      end loop ;
    end if ;
  end resoudre ;

```



```

        return valeur ;
    end resoudre ;
end package body troisetats ;

```

Le traitement des sorties d'impédance intermédiaire, où les niveaux haut ou bas sont obtenus *via* une résistance de rappel (*pull up* ou *pull down*), peut se faire en suivant le même principe. Le nombre de combinaisons à traiter est simplement plus élevé.

Insistons sur la remarque introductive de ce paragraphe consacré à la modélisation, l'exemple qui précède n'est pas synthétisé correctement. Il manque une indication<sup>1</sup> qui informe le synthétiseur du caractère particulier attaché à notre type `x_0_1_z`, qui ne doit pas être traité comme un type énuméré ordinaire, comme ceux que nous avons rencontrés dans la description de machines d'états, par exemple.

### c) Signaux gardés

Les signaux gardés doivent être résolus, leur fonction de résolution retourne une valeur par défaut si toutes ses sources sont déconnectées. Cette valeur par défaut est utilisée pour modéliser l'état d'un signal de catégorie bus laissé « en l'air ». Un signal gardé de catégorie `register` dont tous les pilotes sont déconnectés conserve la valeur qui précédait la dernière déconnexion.

## 2.7.4 Les paquetages de la librairie IEEE<sup>2</sup>

La modélisation des bus, des opérateurs logiques câblés et autres objets qui échappent au monde des valeurs binaires classiques, a donné lieu à une floraison de types « maison », liés à un outil de développement particulier. Il en a été de même pour les fonctions de manipulation des vecteurs d'objets binaires ou des types précédents. La portabilité des programmes sources n'étant plus assurée, particulièrement en synthèse, un effort de standardisation était indispensable.

La librairie IEEE est le fruit de cette standardisation. Pratiquement généralisée, elle est aussi indissociable d'un compilateur VHDL que la librairie STD elle-même.

### a) Des types logiques simples multivalués

Un signal logique élémentaire (une seule équipotentielle du schéma) est défini comme pouvant prendre neuf valeurs (type énuméré `std_ulogic`), dont certaines n'ont de sens que pour la simulation :

- 'U' représente une équipotentielle non initialisée. Par défaut tous les signaux sont initialisés à cette valeur, ce qui permet de repérer les lacunes dans la couverture d'un programme de vérification : les nœuds du montage qui conservent cette valeur n'ont pas été atteints par les tests. N'est pas pris en compte en synthèse.

1. Dans le monde pré-IEEE certains attributs « magiques » permettaient parfois de piloter le synthétiseur vers la sémantique souhaitée.

2. Institute of Electrical and Electronics Engineers.

- 'X' représente le résultat d'un conflit *fort* (*strong unknown*) entre deux sorties standards. Son apparition en simulation représente un cas de mauvais fonctionnement du montage. N'est pas pris en compte en synthèse <sup>1</sup>.
- '0' représente l'état logique bas *fort* d'une sortie standard. L'outil de synthèse traite de la même façon cette valeur, son équivalent du type bit et la valeur booléenne FALSE.
- '1' représente l'état logique haut *fort* d'une sortie standard. L'outil de synthèse traite de la même façon cette valeur, son équivalent du type bit et la valeur booléenne TRUE.
- 'Z' représente l'état haute impédance d'une sortie trois états. Prise en compte par l'outil de synthèse pour générer les portes correspondantes avec leur commande :  
if oe = ... then s <= 'Z' ...
- 'W' représente le résultat d'un conflit *faible* (*weak unknown*), entre deux sorties « résistives ». Il est dominé par un niveau *fort*. Son apparition en simulation représente un cas de mauvais fonctionnement du montage. N'est pas pris en compte en synthèse.
- 'L' représente l'état logique bas *faible* d'une sortie résistive (*pull down*). Il est dominé par un niveau *fort*. L'outil de synthèse peut le traiter de la même façon que '0', si la technologie du circuit cible ne distingue pas les deux forces.
- 'H' représente l'état logique haut *faible* d'une sortie résistive (*pull up*). Il est dominé par un niveau *fort*. L'outil de synthèse peut le traiter de la même façon que '1', si la technologie du circuit cible ne distingue pas les deux forces.
- '-' représente un niveau logique indifférent. Sans signification en synthèse, il peut être utilisé pour simplifier des opérateurs logiques.

#### ► Les types std\_ulogic et std\_logic

Le paquetage ieee.std\_logic\_1164 définit le type std\_ulogic qui est le type de base de la librairie IEEE :

```
type std_ulogic is ( 'U', -- Uninitialized
                    'X', -- Forcing Unknown
                    '0', -- Forcing 0
                    '1', -- Forcing 1
                    'Z', -- High Impedance
                    'W', -- Weak Unknown
                    'L', -- Weak 0
                    'H', -- Weak 1
                    '-' -- Don't care
                    );
```

1. Le problème de la synthèse de ces niveaux virtuels se pose, par exemple, quand ils apparaissent dans des tests : if a = 'X' then ... L'outil de synthèse doit considérer que le résultat du test est toujours faux puisque de telles valeurs ne peuvent pas apparaître dans le vrai circuit. Une bonne pratique consiste à ne pas inclure de telles instructions au sein de modules synthétisables, mais de les réserver aux modules de test pur.

Le sous-type `std_logic`, qui est le plus utilisé, est associé à la fonction de résolution `resolved` :

```
function resolved ( s : std_ulogic_vector )
    return std_ulogic;
subtype std_logic is resolved std_ulogic;
```

Cette fonction de résolution utilise une table de gestion des conflits qui reproduit les forces respectives des valeurs du type :

```
type stdlogic_table is array(std_ulogic, std_ulogic)
    of std_ulogic;
constant resolution_table : stdlogic_table := (
-----
--| U   X   0   1   Z   W   L   H   -   | |
-----
( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
( 'U', 'X', '0', 'X', '0', '0', '0', '0', '0', 'X' ), -- | 0 |
( 'U', 'X', 'X', '1', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | - |
);
```

Le paquetage définit également des vecteurs :

```
type std_logic_vector is array ( natural range <> )
    of std_logic;

type std_ulogic_vector is array ( natural range <> )
    of std_ulogic;
```

Et les sous-types résolus X01, X01Z, UX01 et UX01Z.

Des fonctions de conversions permettent de passer du type binaire aux types IEEE et réciproquement, ou d'un type IEEE à l'autre :

```
function To_bit ( s : std_ulogic; xmap : bit := '0' )
    return bit;
function To_bitvector ( s : std_logic_vector ;
    xmap : bit := '0' ) return bit_vector;
function To_bitvector ( s : std_ulogic_vector;
    xmap : bit := '0' ) return bit_vector;
function To_StdULogic ( b : bit ) return std_ulogic;
function To_StdLogicVector ( b : bit_vector )
    return std_logic_vector;
function To_StdLogicVector ( s : std_ulogic_vector )
    return std_logic_vector;
function To_StdULogicVector ( b : bit_vector )
    return std_ulogic_vector;
```

Par défaut les fonctions comme `To_bit` remplacent, au moyen du paramètre `xmap`, toutes les valeurs autres que '1' et 'H' par '0'.

La détection d'un front d'horloge se fait au moyen des fonctions :

```
function rising_edge (signal s : std_ulogic) return boolean;
function falling_edge(signal s : std_ulogic) return boolean;
```

Tous les opérateurs logiques sont surchargés pour agir sur les types IEEE comme sur les types binaires. Ces opérateurs retournent les valeurs *fortes* '0', '1', 'X', ou 'U'.

### ➤ Nombres et vecteurs

Un nombre entier peut être assimilé à un vecteur, dont les éléments sont les coefficients binaires de son développement polynomial en base deux. Restent à définir sur ces objets les opérateurs arithmétiques, ce que permet la surcharge d'opérateurs.

Les paquetages `numeric_std` et `numeric_bit` correspondent à l'utilisation, sous forme de nombres, de vecteurs dont les éléments sont des types `std_logic` et `bit`, respectivement. Comme les types définis dans ces paquetages portent les mêmes noms, ils ne peuvent pas être rendus visibles simultanément dans un même module de programme ; il faut choisir un contexte ou l'autre.

Les deux paquetages ont pratiquement la même structure, et définissent les types `signed` et `unsigned` :

```
-- ieee.numeric_bit :
type UNSIGNED is array (NATURAL range <> ) of BIT;
type SIGNED is array (NATURAL range <> ) of BIT;

-- ieee.numeric_std :
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```

Les vecteurs doivent être rangés dans l'ordre descendant (`downto`) de l'indice, de sorte que le coefficient de poids fort soit toujours écrit à gauche, et que le coefficient de poids faible, d'indice 0, soit à droite, ce qui est l'ordre naturel. La représentation interne des nombres signés correspond au code complément à deux, dans laquelle le chiffre de poids fort est le bit de signe ('1' pour un nombre négatif, '0' pour un nombre positif ou nul).

Les opérations prédéfinies dans ces paquetages, agissant sur les types `signed` et `unsigned`, sont :

- Les opérations arithmétiques.
- Les comparaisons.
- Les opérations logiques pour `numeric_std`, elles sont natives pour les vecteurs de bits.
- Des fonctions de conversion entre nombres et vecteurs :

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
function TO_INTEGER (ARG: SIGNED) return INTEGER;
function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
```

```
function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL)
    return SIGNED;
```

- Une fonction de recherche d'identité (`std_match`) qui utilise l'état *don't care* du type `std_logic` comme joker.
- Les recherches de fronts (`rising_edge` et `falling_edge`), agissant sur le type `bit`, dans `numeric_bit`.

### b) Des opérations prédéfinies

Les opérandes et les résultats des opérateurs arithmétiques et relationnels appellent quelques commentaires. Ces opérateurs acceptent comme opérandes deux vecteurs ou un vecteur et un nombre. Dans le cas de deux vecteurs dont l'un est signé, l'autre pas, il est à la charge du programmeur de prévoir les conversions de types nécessaires.

#### » Les opérateurs arithmétiques

L'addition et la soustraction sont faites sans aucun test de débordement, ni génération de retenue finale. La dimension du résultat est celle du plus grand des opérandes, quand l'opération porte sur deux vecteurs, ou celle du vecteur passé en argument dans le cas d'une opération entre un vecteur et un nombre. Le résultat est donc calculé implicitement modulo  $2^n$ , où  $n$  est la dimension du vecteur retourné. Ce modulo implicite allège, par exemple, la description d'un compteur binaire, évitant au programmeur de prévoir explicitement l'incrémement modulo la taille du compteur<sup>1</sup>.

La multiplication retourne un résultat dont la dimension est calculée pour pouvoir contenir le plus grand résultat possible : somme des dimensions des opérandes moins un, dans le cas de deux vecteurs, double de la dimension du vecteur passé en paramètre moins un dans le cas de la multiplication d'un vecteur par un nombre.

Pour la division entre deux vecteurs, le quotient a la dimension du dividende, le reste celle du diviseur. Quand les opérations portent sur un nombre et un vecteur, la dimension du résultat ne peut pas dépasser celle du vecteur, que celui-ci soit dividende ou diviseur.

#### » Les opérateurs relationnels

Quand on compare des vecteurs interprétés comme étant des nombres, les résultats peuvent être différents de ceux que l'on obtiendrait en comparant des vecteurs sans signification. Le tableau ci-après donne quelques exemples de résultats en fonction des types des opérandes.

Ces résultats se comprennent aisément si on garde à l'esprit que la comparaison de vecteurs ordinaires, sans signification numérique, se fait de gauche à droite sans notion de poids attaché aux éléments binaires.

1. Cette simplification disparaît évidemment dans le cas d'un compteur dont la longueur du cycle n'est pas fixée par une puissance de 2.

Expression	Types des opérandes		
	bit_vector	unsigned	signed
"001" = "00001"	FALSE	TRUE	TRUE
"001" > "00001"	TRUE	FALSE	FALSE
"100" < "01000"	FALSE	TRUE	TRUE
"010" < "10000"	TRUE	TRUE	FALSE
"100" < "00100"	FALSE	FALSE	TRUE

### ► Compteur décimal

Comme illustration de l'utilisation de la librairie IEEE, donnons le code source d'une version possible de la décade, instanciée comme composant dans un compteur décimal :

```

library ieee ;
use ieee.numeric_bit.all ;

ARCHITECTURE vecteur OF decade IS
    signal countTemp: unsigned(3 downto 0) ;
begin
    count <= chiffre(to_integer(countTemp)) ; -- conversion
    dix <= en when countTemp = 9 else '0' ;
    incre : PROCESS
        BEGIN
            WAIT UNTIL rising_edge(clk) ;
            if raz = '1' then
                countTemp <= X"0" ;
            -- ou :    countTemp <= to_unsigned(0) ;
            elsif en = '1' then
                countTemp <= (countTemp + 1) mod 10 ;
            end if ;
        END process incre ;
    END vecteur ;

```

L'intérêt de ce programme réside dans l'aspect évident des choses, le signal countTemp, un vecteur d'éléments binaires, est manipulé dans des opérations arithmétiques exactement comme s'il s'agissait d'un nombre. Seules certaines opérations de conversions rappellent les différences de nature entre les types unsigned et integer.

### 2.7.5 Des outils de simulation

À quelques exceptions près, principalement rencontrées à l'occasion d'exemples, tous les éléments du langage que nous avons abordé jusqu'à présent étaient, dans leur principe au moins, synthétisables. La frontière entre ce qui est synthétisable et ce qui ne l'est pas n'est pas une ligne bien nette, nous avons déjà signalé ce point.

Les éléments que nous abordons dans la suite ne présentent pas d'ambiguïté, ce sont des outils d'élaboration et de modélisation, ils ne laissent aucune trace physique dans un circuit. De nombreux outils de synthèse ne tolèrent d'ailleurs pas leur présence dans un programme source, d'autres, dotés d'une plus grande intelligence, les ignorent, si c'est possible. Une règle d'or consiste à ne jamais émailler une unité de conception destinée à créer un circuit, entité ou configuration et paquetages associés, d'instructions non synthétisables.

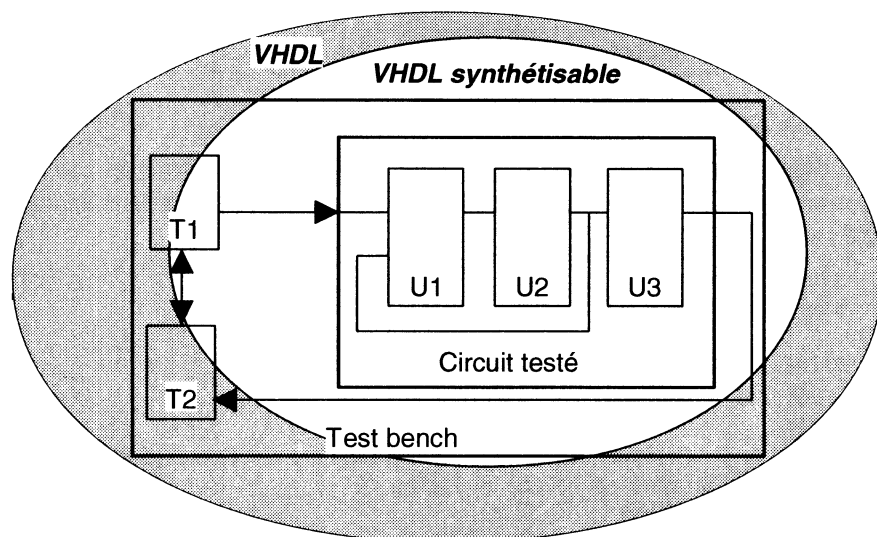


Figure 2-26 Test et synthèse.

La figure 2-26 illustre cette règle : on souhaite vérifier le fonctionnement du modèle synthétisable d'un circuit constitué de trois blocs U1, U2 et U3. Pour ce faire le circuit est inclus dans un module de test, ou *test bench*, qui utilise lui même deux autres blocs, T1 et T2. Le circuit lui-même et ses constituants ne contiennent que du code synthétisable, les modules de test, entité supérieure comprise, peuvent contenir des instructions synthétisables, d'autres non. Ils dialoguent avec le module testé *via* des signaux, mais en aucun cas les instructions de test elles-mêmes ne sont incluses dans la description du circuit.

#### a) Les fichiers

VHDL n'est pas un langage de programmation au sens usuel du terme, ses possibilités en termes de manipulations de fichiers sont donc réduites au strict minimum : ouverture (en lecture, en écriture, ou en ajout), fermeture, lecture ou écriture séquentielle et détection de la fin du fichier. La manipulation des fichiers a l'avantage d'être extrêmement simple.

Les applications typiques des fichiers sont la lecture de données initiales, pour la simulation ou la synthèse, et l'écriture de résultats de simulation à des fins d'analyse

par un autre outil logiciel : analyse de Fourier de la sortie d'un opérateur de traitement de signal, pilotage d'un équipement de test.

### ➤ Déclaration et ouverture

Le mot `file` représente à la fois un type et une classe d'objets. Commençons par quelques exemples :

```
type int_fich is file of integer ;
```

Définit un type de fichier binaire, qui est une séquence, de longueur arbitraire, de nombres entiers.

```
file f1 : int_fich ;
```

Déclare un objet `f1` de classe `file` et de type `integer_file`. La correspondance entre le nom `f1` et un fichier physique du système d'exploitation sera faite à l'exécution du programme par l'appel explicite à la procédure `file_open`. Il est possible d'ouvrir un fichier dès sa déclaration :

```
file f2 : int_fich is "donnees.txt" ;
```

Déclare un objet `f2` de type `int_fich` et l'associe au fichier de nom `donnees.txt`, ouvert en lecture (paramètre par défaut), en faisant un appel implicite `file_open(f2,"donnees.txt")`.

```
file f3 : int_fich open write_mode is "resultat.txt" ;
```

Déclare un objet `f3` de type `int_fich` et l'associe au fichier de nom `resultat.txt`, qui est ouvert en écriture, en faisant un appel implicite `file_open(f2,"resultat.txt",write_mode)`.

La syntaxe de déclaration d'un fichier est :

```
file liste_noms : type_fic [ouverture] ;
ouverture ::=
  [open file_open_kind_expression] is nom_système
```

Les modes d'ouverture possibles sont déclarés dans le paquetage standard :

```
type file_open_kind is (
  read_mode,
  write_mode,
  append_mode) ;
```

En l'absence de précision contraire, le fichier est ouvert en mode lecture. Les modes `write` et `append` diffèrent quand le fichier existe avant l'ouverture, dans le premier mode l'ancien fichier est détruit et recréé vide, dans le second les nouvelles données seront rajoutées à la fin du fichier existant. Il est impossible de déclarer un même fichier à la fois en lecture et en écriture, de toute façon le parallélisme du langage rendrait une telle opération extrêmement scabreuse.

Le type du fichier doit avoir été défini :

```
type type_fic is file of type_elt ;
```

Les éléments peuvent être de types scalaire, enregistrement ou tableaux contraints (de dimensions connues), cela exclut notamment les pointeurs et un autre type



fichier. La déclaration d'un type de fichier déclare implicitement les procédures d'ouverture, de fermeture, de lecture, d'écriture et la fonction de détection de fin de fichier :

```

procedure file_open (file F : type_fic ;
                    External_Name : in string ;
                    Open_Kind : in file_open_kind := read_mode) ;
procedure file_open (Status : out file_open_status ;
                    file F : type_fic ;
                    External_Name : in string ;
                    Open_Kind : in file_open_kind := read_mode) ;
procedure file_close (file F : type_fic) ;
procedure read (file F : type_fic ; VALUE : out type_elt) ;
procedure write (file F : type_fic ; VALUE : in type_elt) ;
function endfile (file F : type_fic) return boolean ;

```

Les valeurs du paramètre Status, code de retour éventuel de la procédure file\_open, sont déclarées dans le paquetage standard :

```

type file_open_status is (open_ok, status_error,
                        name_error, mode_error);

```

Ces déclarations implicites de procédures appellent deux remarques :

- Les fichiers VHDL sont structurés, ce qui simplifie notablement leurs manipulations. Point n'est besoin d'utiliser des spécificateurs de formats, le type de fichier renseigne sur sa structure, les types des arguments des procédures read et write les renseignent sur les formats des données à extraire ou inscrire.
- Les déclarations implicites précédentes peuvent provoquer des conflits de noms difficiles à localiser si les noms read ou write sont utilisés par ailleurs<sup>1</sup>, bien qu'il ne s'agisse pas de mots réservés au sens strict, un bon réflexe est de ne pas les employer pour d'autres usages.

### » Le paquetage textio

Quand rien de précis ne s'y oppose, l'emploi de fichiers textes est de loin préférable à celui de fichiers binaires. Le seul inconvénient est un volume et un temps d'accès, lié aux conversions nécessaires, plus importants. L'avantage immense est la lisibilité des fichiers au moyen de n'importe quel éditeur de texte, ce qui facilite considérablement la mise au point des programmes<sup>2</sup>. Le format texte n'empêche évidemment pas de consigner dans un fichier des données numériques, simplement ces dernières y sont inscrites sous une forme humainement lisible et converties en binaire dans le programme utilisateur.

Le paquetage textio, de la librairie std, définit sémantiquement un fichier texte comme organisé en lignes de longueurs variables. L'accès se fait en deux temps :

1. Histoire vraie : un programme utilisait ces noms comme signaux de commandes de mémoires. Tout allait bien jusqu'au moment où le programmeur voulut créer un fichier.
2. Cette remarque n'est pas spécifique à VHDL, en phase de mise au point d'un programme il est toujours bien plus confortable de manipuler des fichiers textes, quitte à passer au format binaire une fois le programme déverminé.

accès à une ligne entière (chaîne de caractère terminée par un marqueur de fin de ligne<sup>1</sup>) qui est stockée dans une zone mémoire allouée dynamiquement au moyen d'un pointeur, exploitation ligne par ligne du contenu. La figure 2-27 résume le principe.

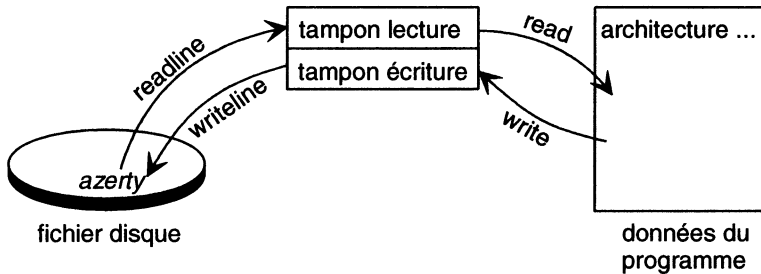


Figure 2-27 Fichiers textes.

Quelques déclarations utiles données dans `textio` :

```

-- type pointeur adapté aux tampons :
type LINE is access string ;

-- type de fichier texte :
type TEXT is file of string ;

-- procédures d'échanges fichier <-> tampons :
procedure READLINE(file f: TEXT; L: out LINE) ;
procedure WRITELINE(file f: TEXT; L: inout LINE) ;

-- quelques procédures de lecture de données du tampon :
procedure READ(L:inout LINE; VALUE: out bit_vector) ;
procedure READ(L:inout LINE; VALUE: out bit_vector ;
               GOOD : out BOOLEAN);
procedure READ(L:inout LINE; VALUE: out integer);
procedure READ(L:inout LINE; VALUE: out integer;
               GOOD : out BOOLEAN);

-- etc

-- quelques procédures d'écriture dans le tampon :
subtype WIDTH is natural;    -- largeur du champ
type SIDE is (right, left) ; -- pour le cadrage
procedure WRITE(L: inout LINE; VALUE: in bit_vector;
               JUSTIFIED: in SIDE := right;
               FIELD: in WIDTH := 0);
procedure WRITE(L: inout LINE; VALUE: in integer;

```

1. *New line* ou *carriage return*, suivant les systèmes. En principe ces choses sont transparentes pour le programmeur... sauf s'il transporte des fichiers d'un système à un autre. Là encore, rien qui soit lié à VHDL.

```
JUSTIFIED: in SIDE := right;
FIELD: in WIDTH := 0);
```

Les procédures `read` et `write` sont surchargées en fonction des types des données échangées. Pour chaque type deux versions de `read` sont fournies : sans et avec contrôle d'erreur (de format, par exemple) au moyen d'un paramètre booléen supplémentaire. Les procédures `write` permettent de contrôler les largeurs de champs et la justification dans un champ. Les valeurs par défaut des paramètres associés assurent un fonctionnement « automatique », adapté aux données.

Les accès aux tampons sont séquentiels, toute donnée extraite d'un tampon de lecture positionne le pointeur sur la prochaine donnée à lire, toute donnée rajoutée à un tampon d'écriture l'est à la fin de la chaîne de caractères correspondante.

Le programme ci-dessous lit un fichier de données qui contient des valeurs entières comprises entre 0 et 255, une par ligne. S'il n'y a pas d'erreur de lecture, ces données sont chargées dans un vecteur de huit octets, qui sert à construire un modèle rudimentaire de mémoire ROM :

```
library ieee ;
use ieee.std_logic_1164.all ;

package rompkg is
    constant ad_nb_bits : integer := 3 ; -- 3 bits d'adresse
    constant taille : integer := 2**ad_nb_bits ; -- 8 octets
    subtype octet is std_logic_vector(7 downto 0) ;
    type rom_tbl is array(natural range <>) of octet ;
end package rompkg ;

library ieee ;
use ieee.std_logic_1164.all, ieee.numeric_std.all ;
use std.textio.all, work.rompkg.all ;

entity rom is
    port(ce : in std_logic ; -- commande trois états
         adresse : in unsigned(ad_nb_bits - 1 downto 0) ;
         donnee : out std_logic_vector(7 downto 0) ) ;
end rom ;

architecture fichier of rom is
    signal rom_val : rom_tbl(0 to taille - 1) ;
begin
    lecture : process
        file donnees : text is "contenu.txt" ;
        variable ligne,message : line ;
        variable i : natural := 0 ;
        variable rom_buf : integer range 0 to 255 ;
        variable ok : boolean ;
    begin
        while not endfile(donnees) loop
            readline(donnees,ligne);
            read(ligne,rom_buf,ok) ;
```

```

if ok then -- pas d'erreur de format
    rom_val(i) <= octet(to_unsigned(rom_buf,8)) ;
else
    -- erreur de format
    write(message,string'(" !!! format !!! : ") ;
    write(message, ligne.all) ;
    write(message,string'(" à la ligne numero ") ;
    write(message,i) ; -- numéro de la ligne fausse
    report message.all ; -- affichage console
end if ;
if i < taille - 1 then
    i := i + 1 ;
else
    report "rom pleine" ; -- affichage console
    exit ; -- sortie de boucle
end if ;
end loop ;
wait ; -- fin d'initialisation de la mémoire
end process lecture ;
donnee <= rom_val(to_integer(adresse)) when ce = '0'
    else (others => 'Z') ;
end fichier ;

```

Dans le programme précédent on utilise la procédure `write` pour construire le message d'erreur qui sera affiché à la console (par l'instruction `report` que nous reverrons), si le fichier de données contient une valeur dont le format est erroné. L'appel à cette procédure permet de concaténer des chaînes de caractères et d'y inclure le numéro de ligne faux sous forme de texte.

### ➤ Les entrées sorties standards

Le paquetage `textio` contient également les déclarations qui permettent de dialoguer avec la console de l'utilisateur : l'entrée et la sortie standard, généralement le clavier et l'écran, nommées respectivement `input` et `output`. Ces deux fichiers texte sont ouverts dès qu'est rendu visible ce paquetage par la clause `use` :

```

file input : TEXT open read_mode is "STD_INPUT";
file output : TEXT open write_mode is "STD_OUTPUT";

```

On peut, par exemple, modifier le programme précédent pour demander à l'utilisateur le nom du fichier de données à utiliser pour initialiser la mémoire :

```

lecture : process
    variable nom_fic : line ;
    variable openok : file_open_status ;
    variable ligne,message : line ;
    -- Mêmes déclarations que précédemment
begin
    write(message,string'("fichier de données ?")) ;
    writeline(output,message) ;
    readline(input,nom_fic) ;
    file_open(openok,donnees, nom_fic.all, read_mode) ;
    if openok /= OPEN_OK then

```

```

        report "erreur ouverture" ; -- affichage console
        wait ;
    end if ;
    while not endfile(donnees) loop
        -- Le reste sans changement.
    
```

Les détails qui régissent les affichages des différents messages à la console dépendent du système utilisé. Dans les environnements multifenêtres input et output correspondent généralement à une fenêtre de dialogue qui s'ouvre quand cela est nécessaire.

Les exemples qui précèdent nous ont, en passant, donné l'occasion d'illustrer par quelques exemples les manipulations de pointeurs. Le contenu des objets auquel on accède par un pointeur s'obtient par un nom composé, comme dans `file_open(openok,donnees, nom_fic.all, read_mode)`.

### b) Le traitement des fautes

Un modèle de circuit est destiné, entre autres, à détecter les erreurs de conception, à trouver les limites de fonctionnement du circuit, à comparer la conformité entre une version comportementale et sa réalisation structurelle, etc. Quand une faute se présente le modèle doit avoir le moyen d'en informer l'utilisateur par des messages et, éventuellement, de provoquer une réaction du simulateur, un point d'arrêt, par exemple. Les deux instructions qui suivent rentrent dans ce contexte.

#### » Tester une condition : assert

L'instruction `assert`, dont les versions concurrente et séquentielle existent, sert à dénoncer le niveau de gravité (*severity level*) d'une faute et à indiquer quelle en est l'origine, en désignant l'unité de conception qui a découvert la faute.

```

[label :] assert condition
        [report message_texte]
        [severity severity_level_expression] ;
    
```

Si la condition est fausse un message est affiché et le niveau de gravité annoncé au simulateur. Le message affiché contient le niveau de gravité, le texte prévu par l'utilisateur, l'instant où l'erreur a été détectée et le nom de l'unité de conception qui a découvert la faute.

Le niveau de gravité appartient à l'une des valeurs définies dans le paquetage standard :

```

type severity_level is (note, warning, error, failure) ;
    
```

Si la clause `severity` est absente, le niveau implicite est `error`. La configuration d'un simulateur permet d'arrêter la simulation dès qu'un niveau de gravité spécifié est atteint, typiquement `failure` par défaut.

L'exemple qui suit décrit un espion de bus de contrôle, plus précisément, il prévient dès qu'une erreur de décodage risque de créer un conflit de bus. Pour ce faire, il vérifie qu'il n'y a jamais plus d'un '0' dans un vecteur de signaux binaires surveillé, et annonce un avertissement (`warning`) si la condition est fausse :

```

entity espion is
    generic(taille : integer := 8) ;
    port(observe : in bit_vector(0 to taille - 1) ) ;
end espion ;

architecture decode_ok of espion is
begin
    regarde : process (observe)
        variable test : integer range 0 to taille ;
    begin
        test := 0 ;
        for i in observe'range loop
            if observe(i) = '0' then
                test := test + 1 ;
            end if ;
        end loop ;
        assert test <= 1 -- comparaison
            report "erreur de décodage"
            severity warning ;
    end process regarde ;
end decode_ok ;

```

Une architecture de test instancie l'espion précédent comme composant :

```

architecture test of espi_tst is
    component espion is
        generic(taille : integer := 8) ;
        port(observe : in bit_vector(0 to taille - 1) ) ;
    end component espion ;
    signal decode : bit_vector(0 to 7) := X"FF" ; -- ok
begin
    surveille : espion generic map(8)
        port map(decode) ;
    decode <= X"FE" after 10 ns, -- ok
            X"7E" after 20 ns, -- erreur
            X"FF" after 30 ns, -- ok
            X"00" after 100 ns ; -- erreur
end test ;

```

À la simulation les messages obtenus confirment les erreurs :

```

** Warning: erreur de décodage
   Time: 20 ns  Iteration: 0  Instance:/surveille
** Warning: erreur de décodage
   Time: 100 ns Iteration: 0  Instance:/surveille

```

L'architecture de l'espion ne contient que du code passif, c'est-à-dire qui ne modifie la valeur d'aucun signal. Un tel programme passif peut être mis directement dans la zone d'instructions de l'entité :

```

entity espion is
    generic(taille : integer := 8) ;
    port(observe : in bit_vector(0 to taille - 1) ) ;
begin

```

```

regarde : process (observe)
    variable test : integer range 0 to taille ;
begin
    test := 0 ;
    for i in observe'range loop
        if observe(i) = '0' then
            test := test + 1 ;
        end if ;
    end loop ;
    assert test <= 1
        report "erreur de décodage"
        severity warning ;
end process regarde ;
end espion ;

```

Sans intérêt ici, puisque l'architecture deviendrait vide (elle doit exister), cette méthode permet parfois de bien séparer les parties comportement et traitement d'erreurs des modèles.

#### ➤ Signaler une faute : report

Le test de la condition d'erreur peut être fait par le programme, l'instruction `report` permettant de signaler la faute :

```

[label :] report message_texte
        [severity severity_level_expression] ;

```

La gravité par défaut de l'erreur est ici une simple note.

Une variante du programme espion serait alors :

```

if test > 1 then
    report "erreur de décodage"
    severity warning ;
end if ;

```

#### c) La gestion du temps simulateur

Le temps du simulateur est une variable de type `time`, qui est initialisée à 0 au lancement du simulateur, et qui ne peut pas décroître.

#### ➤ La fonction now

Le paquetage standard définit une fonction `now`, qui retourne la valeur du temps actuel :

```

subtype delay_length is time range 0 fs to time'high;
impure function now return delay_length;

```

Cette fonction n'a pas de paramètre d'appel. Le qualificatif `impure` de la déclaration indique une fonction qui retourne une valeur qui dépend d'autre chose que des arguments d'appel, cas évident en l'occurrence<sup>1</sup>. Le temps retourné par `now` est le temps absolu du simulateur.

### ► Formes d'ondes projetées

Nous avons eu l'occasion de rencontrer de nombreux exemples de formes d'ondes projetées, comme opérandes d'instructions d'affectation. Nous nous contenterons ici de préciser quelques points.

Une forme d'onde définit une séquence de valeurs prévues pour un signal dans le futur :

```

    waveform ::=
        waveform_element {, waveform_element}
    | unaffected
    waveform_element ::=
        value_expression [ after time_expression ]
    | null [ after time_expression ]

```

Les valeurs des temps indiquées, classées dans un ordre croissant, ne doivent jamais être négatives; elles se rapportent à l'instant présent (que retournerait un appel à `now`), instant où l'instruction d'affectation de signal est activée, elles définissent des événements à venir. La valeur de signal `null` indique une déconnexion du pilote du signal, applicable aux signaux gardés uniquement.

### d) Les retards dans les circuits

Tout opérateur logique réel présente un retard : un événement sur une entrée provoque la modification d'une sortie après un certain laps de temps. La modélisation de ces retards utilise les formes d'ondes projetées. Un inverseur rudimentaire peut, par exemple, être représenté par le programme :

```

    library ieee ;
    use ieee.std_logic_1164.all ;

    entity als_04 is
        generic(tplh : time := 11 ns ;
               tphl : time := 8 ns ) ;
        port(a : in std_logic ;
             y : out std_logic ) ;
    end als_04 ;

    architecture retard of als_04 is
    begin
        y <= '1' after tplh when a = '0'
            else '0' after tphl ;
    end retard ;

```

Une telle modélisation naïve n'est évidemment guère généralisable, elle n'assure, en particulier, aucune indépendance entre la fonction logique et la modélisation des retards. La création d'une procédure générale de modélisation des retards permet de rendre indépendantes la description logique et la modélisation temporelle :

1. Les fonctions ordinaires sont pures. Des fonctions impures peuvent être créées, par exemple, dans d'autres langages (attribut `foreign`) ou par l'appel d'autres fonctions impures. Leur emploi nuit à la portabilité des programmes, sauf pour celles des bibliothèques standard.



```

library ieee ;
use ieee.std_logic_1164.all ;

package temporel is
    procedure propagat (signal sortie : out std_logic ;
                        variable nouvelle : in std_logic ;
                        variable ancienne : inout std_logic ;
                        constant t1h,t1l : in time) ;
end package temporel ;

package body temporel is
    procedure propagat (signal sortie : out std_logic ;
                        variable nouvelle : in std_logic ;
                        variable ancienne : inout std_logic ;
                        constant t1h,t1l : in time) is
    begin
        case std_logic_vector'(ancienne,nouvelle) is
            when "00" | "11" => return ;
            when "01" | "X1" => sortie <= nouvelle after t1h ;
                                ancienne := nouvelle ;
            when "10" | "X0" => sortie <= nouvelle after t1l ;
                                ancienne := nouvelle ;
            when others => sortie <= 'X' ;
                                ancienne := 'X' ;
        end case ;
    end procedure propagat ;
end package body temporel ;

```

L'inverseur devient :

```

use work.temporel.all ;

architecture generale of als_04 is
begin
    inv : process (a)
        variable futurs, anciens : std_logic ;
    begin
        -- fonctionnement logique idéal :
        futurs := not a ;
        -- transaction avec retards :
        propagat(sortie => y,
                 nouvelle => futurs,
                 ancienne => anciens,
                 t1h => t1h, t1l => t1l) ;
    end process inv ;
end generale ;

```

La même structure d'architecture peut servir à modéliser n'importe quel opérateur combinatoire à une sortie, quelle que soit son équation logique. C'est, en très simplifiée, la logique de la démarche adoptée pour la librairie VITAL (*Vhdl Initiative Towards Asics Libraries*), qui est un standard de fait dans le monde de la modélisation des ASICs et FPGAs ; nous en reparlerons.

### ► Mode inertiel et mode transport

Le modèle d'un circuit complexe génère des impulsions très étroites, dues aux différences de temps de propagations entre les signaux qui suivent des chemins distincts. Ces impulsions ne correspondent pas forcément à la réalité : un opérateur présente une inertie d'autant plus grande que son temps de propagation est grand, il a tendance à « gommer » des parasites très courts.

La capacité à gommer ou non des impulsions courtes se signifie par l'option `inertial` ou `transport`, dans une instruction d'affectation de signal. La première option, qui est l'option par défaut, conduit à supprimer de la sortie d'un opérateur toute impulsion plus courte que le retard spécifié dans le premier élément de la forme d'onde ; la seconde option transporte vers la sortie toutes les nuances temporelles. Le programme ci-dessous utilise ces deux options et l'option par défaut :

```
architecture exemple of inertie is
  signal e, sinerte, stransport, sdefault : bit ;
begin
  e <= '1' after 20 ns,
       '0' after 40 ns,
       '1' after 60 ns,
       '0' after 65 ns ;
  sinerte    <= inertial not e after 10 ns ;
  stransport <= transport not e after 10 ns ;
  sdefault   <= not e after 10 ns ;
end exemple ;
```

La figure 2-28 illustre le résultat.

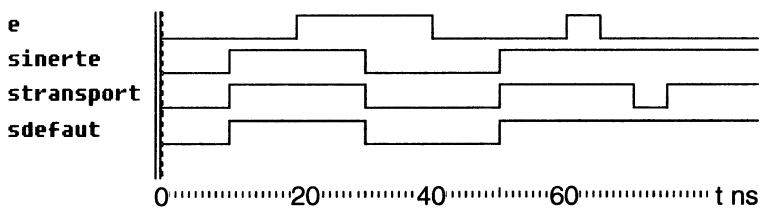


Figure 2-28 Modes inertiel et transport

Les chronogrammes obtenus pour les sorties `sinerte` et `sdefault` ne contiennent pas trace de l'impulsion courte générée au temps 60 ns.

L'option `transport` sert systématiquement à modéliser les retards sur les lignes, qui peuvent être importants dans les systèmes rapides, et qui ne présentent aucun effet d'inertie. La plupart des technologies employées dans les circuits intégrés conduisent plutôt à un modèle utilisant l'option `inertial` pour les opérateurs logiques.

### ► Réjection des impulsions parasites

La norme 1993 du langage a introduit une quantification possible de l'inertie d'un circuit par l'option

```
reject time_expression inertial
```

qui permet de préciser une durée minimum pour les impulsions transmises, indépendante du retard de l'opérateur modélisé. Toute impulsion de durée inférieure à l'expression temporelle sera supprimée. La valeur de cette expression doit, cela se comprend, être inférieure à celle qui apparaît dans le premier élément de la forme d'onde : l'option de réjection sert à établir un continuum entre les options *inertial* et *transport*.

### ► Les violations des règles temporelles

Les signaux appliqués à un circuit logique doivent respecter un certain nombre de contraintes concernant leurs durées et leurs positions respectives : largeurs minimums des impulsions, respect des temps de prépositionnement (*set up time*) et de maintien (*hold time*), pour citer les plus importants. Nous avons présenté, à propos des attributs au paragraphe 2.2.5, un modèle de bascule D qui surveille les violations de *set up* et de *hold*. Ce programme sera beaucoup plus simple à comprendre à la lumière des éléments abordés dans les paragraphes qui précèdent. Les outils principaux de cette modélisation sont les attributs qui décorent les signaux et les instructions de traitement d'erreurs (*assert* et *report*).

Le paquetage `temporel` peut être complété en y rajoutant une procédure relativement générale qui contrôle le bon respect des règles temporelles :

```
package temporel is
  procedure propagat -- identique à la précédente
  procedure test_regles(
    constant nom : in string ;      -- nom entité
    signal clk : in std_logic ;     -- signal d'horloge
    signal entree : in std_logic ;  -- entrée à contrôler
    -- temps écoulé depuis les derniers changements :
    variable clk_front, entree_change : inout time ;
    constant tsu ,th : in time ;
    variable ras : out boolean) ; -- compte rendu
end package temporel ;

package body temporel is
  procedure propagat -- etc.
  procedure test_regles(
    constant nom : in string ;
    signal clk : in std_logic ;
    signal entree : in std_logic ;
    variable clk_front, entree_change : inout time ;
    constant tsu ,th : in time ;
    variable ras : out boolean) is
  begin
    if now = 0 ns then
      clk_front := 0 ns ;
      entree_change := 0 ns ;
      return ;
    end if ;
    ras := true ; -- optimisme initial
```

```

if entree'event then
  if rising_edge(clk) then
    -- traite les changements simultanés
    clk_front := now ;
  end if ;
  if now - clk_front < th then
    report nom & " : violation de thold" ;
    ras := false ;
  end if ;
  entree_change := now ;
elsif rising_edge(clk) then
  if now - entree_change < tsu then
    report nom & " : violation de set up" ;
    ras := false ;
  end if ;
  clk_front := now ;
elsif
  clk'event and (clk = '1' or clk'last_value = '0') then
    report nom & " : transition d'horloge illégale" ;
    ras := false ;
  end if ;
end procedure test_regles ;
end package body temporel ;

```

Par rapport à la version initiale, un certain nombre d'attributs de signaux (*S'stable(...)*, par exemple) ne peuvent plus être employés car ils réclament des arguments statiques, ce qui n'est évidemment pas le cas à l'intérieur d'une procédure qui doit pouvoir être appelée dynamiquement. Le principe est de mémoriser dans l'unité de conception appelante les données temporelles nécessaires aux contrôles. Ces données, que l'on a ici simplifiées au maximum, permettent de connaître le temps écoulé depuis le dernier changement intéressant<sup>1</sup> de chaque signal d'entrée.

Une version plus générale de la bascule D se construit en trois étapes : test du respect des règles, description du fonctionnement idéal si aucune règle n'est enfreinte et propagation des résultats en sortie.

```

use work.temporel.all ;

architecture generale of bascd is

```

1. Par souci de simplicité, la procédure présentée ne traite que le cas d'horloges actives sur fronts montants. Pour les mêmes raisons, la valeur *tho* du temps de maintien doit être strictement positive, le fonctionnement à *tho* nul ou, *a fortiori* négatif, n'est pas correct. Plus précisément : pour *tho* nul, si on applique des stimuli externes (par le simulateur) tels que horloge et entrée d changent simultanément, la valeur de *d* prise en compte est la nouvelle valeur, ce qui est inexact. Si l'entrée *d* provient d'un autre processus synchrone de la même horloge, le fonctionnement est correct même à temps de propagation et temps de maintien nul. Cette difficulté est un classique des simulateurs logiques.

Le temps de prépositionnement *tsu* doit également être strictement positif, mais ce n'est pas cette fois une restriction critiquable.

```

begin
d_ff : process (hor,d)
  variable futurq, ancienq : std_logic ;
  -- pour chaque entrée à surveiller :
  variable timing_d_ok : boolean ;
  variable last_hor_front, last_d_event : time ;
  begin
    -- contrôle des règles temporelles
    test_regles(nom => bascd'path_name,
                clk => hor,
                entree => d,
                clk_front => last_hor_front,
                entree_change => last_d_event,
                tsu => tsu,
                th => tho,
                ras => timing_d_ok) ;
    -- fonctionnement logique idéal :
    if timing_d_ok then
      if rising_edge(hor) then
        futurq := d ;
      end if ;
    else
      futurq := 'X' ;
    end if ;
    -- transaction avec retards :
    propagat(sortie => q,
             nouvelle => futurq,
             ancienne => ancienq,
             tllh => tp, thl => tp) ; -- tplh = tphl
  end process d_ff ;
end generale ;

```

Le programme précédent ne nécessite que des modifications mineures pour modéliser d'autres types de bascules synchrones, ou même des circuits plus complexes tels que compteurs ou registres. À chaque entrée doit être attachée une variable temporelle qui mémorise l'instant de son dernier changement connu, et une variable booléenne de compte rendu, toutes deux mises à jour par la procédure `test_regles`. Le fonctionnement idéal doit rendre compte de la fonction réalisée, naturellement. L'ensemble des traitements peut être rendu plus universel en autorisant, par l'utilisation de procédures surchargées, le remplacement des signaux scalaires par des vecteurs.

### 2.7.6 Rétroannotation

Les données d'entrée d'un compilateur VHDL de synthèse constituent un ensemble de programmes sources qui décrivent le fonctionnement souhaité du circuit. Ce programme est traduit dans un code intermédiaire, qui dépend du logiciel employé, qui est une description semi-structurale du circuit généré, description dans laquelle interviennent des opérateurs logiques, combinatoires ou séquentiels. Pour en arriver

là, une première phase d'optimisation de haut niveau, pas forcément très poussée, a généralement eu lieu : minimisations des équations logiques, regroupements de termes, etc. Ce code intermédiaire est transmis au logiciel de placement-routage (le *fitter*) qui assure l'optimisation finale, fonction du circuit cible, la répartition des blocs logiques dans le circuit, leur programmation et les interconnexions entre ces blocs. Les données de sortie de la chaîne de transformation se présentent sous forme d'un fichier de programmation, si on reste dans le cadre de cette catégorie de cibles.

Reste à savoir si le résultat est conforme aux espérances, reste à évaluer les performances probables du futur circuit. Tâche d'autant plus délicate que le circuit cible est complexe, que ses caractéristiques dynamiques dépendent des choix de placement et de routage. Pour permettre cette évaluation prévisionnelle, les outils de synthèse génèrent, outre des informations chiffrées dans les fichiers de comptes rendus, un modèle VHDL du circuit qui prend en compte les caractéristiques dynamiques de ce dernier. La figure 2-29 résume l'organisation générale des opérations.

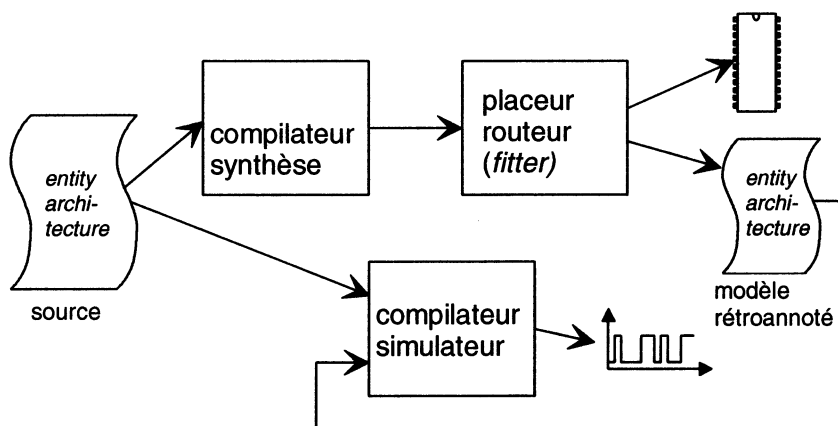


Figure 2-29 Du modèle au circuit, du circuit au modèle.

La génération des informations dynamiques du circuit synthétisé porte le nom de rétroannotation. Le modèle<sup>1</sup> correspondant n'a pas grand-chose à voir avec le programme source d'origine, il s'agit d'une description structurée, qui emploie des composants virtuels, représentés avec leurs retards, et qui protestent en simulation quand les règles temporelles ne sont pas respectées.

1. Non synthétisable ...

### a) Les modèles postsynthèse

Les synthétiseurs utilisent des bibliothèques qui contiennent les descriptions des opérateurs élémentaires des circuits cibles. Le modèle d'une application est obtenu par assemblage de ces opérateurs élémentaires.

Reprenons un exemple simple de bascule D :

```
entity d_edge is
  port (
    hor,d : in bit ;
    s      : out bit
  ) ;
end d_edge ;
architecture essai of d_edge is
  signal q : bit ;
begin
  s <= not q ;
  process (hor)
  begin
    if( hor'event and hor = '1' ) then
      q <= d ;
    end if ;
  end process ;
end essai ;
```

Le programme ci-dessous est obtenu après synthèse de la bascule précédente, dans un PAL 16R8, avec le compilateur WARP de CYPRESS :

```
-- CYPRESS NOVA XVL Structural Architecture
-- JED2VHD Reverse Assembler - 4 IR x61
--   VHDL File: d_edge.vhd
--   Date: Wed Apr 16 10:12:35 1997
-- Disassembly from Jedec file for: c16r8
-- Device Ordercode is: PAL16R8-7JC

library ieee;
use ieee.std_logic_1164.all;

library primitive;
use primitive.c16r8p.all;

ENTITY d_edge IS
  PORT(
    hor : in  std_logic ;
    d   : in  std_logic ;
    s   : inout std_logic ;
    jed_node3 : in std_logic ;
    -- idem pour toutes les broches du circuit
  );
END d_edge;

ARCHITECTURE DSMB of d_edge is
  for all: c20r use entity primitive.c20r (sim);
```

```

SIGNAL one:std_logic:='1';
SIGNAL zero:std_logic:='0';
SIGNAL jed_oept_1:std_logic:='0';
SIGNAL jed_fb_1:std_logic:='0';
--Attribute PIN_NUMBERS of hor: SIGNAL is " 1";
SIGNAL jed_oept_2:std_logic:='0';
--Attribute PIN_NUMBERS of d: SIGNAL is " 2";
SIGNAL jed_oept_11:std_logic:='0';
SIGNAL jed_fb_11:std_logic:='0';
--Attribute PIN_NUMBERS of jed_node11: SIGNAL is "0011";
SIGNAL jed_oept_12:std_logic:='0';
SIGNAL jed_sum_12,jed_yfb_12,jed_qfb_12:std_logic:='0';
--Attribute PIN_NUMBERS of s: SIGNAL is " 12";
BEGIN
Mcell112:c20r
generic map(  7 ns, --tpd paramètres dynamiques
              7 ns, --tea
              7 ns, --ter
              7 ns, --tpzx
              7 ns, --tpxz
              6 ns, --tco
              3 ns, --ts
              0 ns, --th
              3 ns --tw
            )
port map(
    d=>jed_sum_12,
    oe=>jed_node11,
    clk=>hor,
    y=>s,
    qfb=>jed_qfb_12
);
jed_sum_12<= (((d)));
end DSMB;

```

La librairie primitive contient les descriptions comportementales des cellules élémentaires (c20r, par exemple, est une bascule D qui surveille le bon respect des règles temporelles) et, dans le paquetage c16r8p, les déclarations correspondantes.

Dans l'exemple précédent le modèle rétroannoté est lié à une librairie particulière, fournie par un fondeur. Sans gravité pour un circuit très simple, cette absence de normalisation s'est avérée très pénalisante dans le cadre d'applications complexes, d'où l'émergence du standard VITAL.

### b) Vital

Le consortium VITAL a diffusé des librairies de modélisations d'ASICs et de FPGAs qui sont portables et assurent une modélisation suffisamment précise pour offrir une garantie de prévisibilité des résultats. Ces outils de modélisation s'appuient sur une représentation des paramètres dynamiques des circuits compa-



tible avec celle utilisée dans le monde VERILOG, autre langage de description de circuits, à la réputation ancienne et bien établie dans la conception d'ASICs.

La spécification correspondante a évolué pour devenir, en 1995, le standard IEEE 1076.4 (ou VITAL95) qui définit :

- Un paquetage qui contient des procédures de traitement des comportements temporels des circuits, `ieee.vital_timing`.
- Un paquetage qui définit des primitives standards permettant de construire des fonctions combinatoires et séquentielles, `ieee.vital_primitives`.
- Les spécifications des fichiers SDF (*standard delay format*) pour consigner les paramètres dynamiques des unités de conception.
- Les documents de spécification du standard.

L'étude détaillée du standard VITAL sort très largement du cadre de cet ouvrage, l'utilisateur d'un logiciel de synthèse n'ayant pas réellement l'occasion de modéliser lui-même les circuits qu'il emploie. Nous nous contenterons d'en illustrer certains aspects en reprenant l'exemple de notre bascule D, dans une troisième version « compatible VITAL ».

### ► Les primitives

La démarche générale de modélisation d'un opérateur séquentiel correspond à celle présentée<sup>1</sup> à propos de l'architecture générale de notre bascule, celle qui utilise le paquetage temporel :

- Contrôle du bon respect des règles.
- Description de la fonction logique idéale.
- Calcul des sorties en tenant compte des temps de propagation.

Pour décrire le fonctionnement idéal des opérateurs séquentiels, VITAL offre une procédure générale de machine d'états qui utilise comme moteur une table de transitions fournie par l'utilisateur. Le programme ci-dessous conduit au même comportement que les deux précédents, la portabilité en plus :

```
library ieee ;
use ieee.VITAL_Primitives.all;
use ieee.VITAL_Timing.all;

architecture bascd_vital95 of bascd is
begin
    bascule : process (d, hor)
-- données
-- tests de timings
    VARIABLE viol_regles : X01 := '0';
    VARIABLE temps_d : VitalTimingDataType ;

-- équations de fonctionnement : table de transitions
    VARIABLE anciens :
        STD_LOGIC_VECTOR(0 to 2) := (others => 'X');
```

1. Restons modestes, nous nous sommes inspirés de modèles VITAL.

```

variable q_ideal : STD_LOGIC := 'X' ;
CONSTANT transitions :
    VitalStateTableType (1 to 7, 1 to 5) := (
        -----
        --Viol d   hor      q   :   q+
        -----
        ('X', '- ', '- ',      '- ',      'X' ),-- erreur timing
        ('-', '1 ', '/ ',      '- ',      '1' ),-- mise à 1
        ('-', '0 ', '/ ',      '- ',      '0' ),-- mise à 0
        ('-', '- ', '^ ',      '- ',      'X' ),-- fronts hor illégaux:
        ('-', '- ', 'r ',      '- ',      'X' ),-- X->1 et 0->X
        ('-', '- ', 'F ',      '- ',      'S' ),-- fronts descendants
        ('-', '- ', 'S ',      '- ',      'S' ) -- horloge stable
    );

    -- historique de la sortie
    VARIABLE q_histoire : VitalGlitchDataType;
    begin
    -- traitement en trois étapes :

    -- contrôle des règles temporelles :
    VitalSetupHoldCheck (
        TestSignal => d, -- entrée contrôlée
        TestSignalName => "d", -- nom de l'entrée
        RefSignal => hor, -- entrée de référence
        RefSignalName => "hor", -- nom de la référence
        RefTransition => '/', -- transition montante 0->1
        SetupHigh => tsu, -- setup pour d = 1
        SetupLow => tsu, -- setup pour d = 0
        HoldHigh => tho, -- hold pour d = 1
        HoldLow => tho, -- hold pour d = 0
        HeaderMsg => bascd'path_name, -- en-tête message
        MsgSeverity => warning, -- niveau de gravité
        TimingData => temps_d, -- données du traitement
        Violation => viol_regles); -- 'X' si violation

    -- fonctionnement logique idéal :
    VitalStateTable(
        StateTable => transitions, -- table des transitions
        DataIn => (viol_regles, d, hor ), -- entrées
        Result => q_ideal, -- in : état initial; out : final
        PreviousDataIn => anciens); -- entrées précédentes

    -- transaction avec retards :
    VitalPathDelay (
        OutSignal => q, -- signal concerné
        OutSignalName => "q", -- nom du signal
        OutTemp => q_ideal, -- valeur à transporter
        Paths => (0 => (hor'last_event, tp, TRUE)), -- origine
        -- du temps à prendre en compte, chemins multiples
        -- possibles, ici chemin unique.
    );

```

```

        GlitchData => q_histoire); -- mémoire utile pour les
        -- traitements des parasites
    end process;
end bascd_vital95 ;

```

Comme pour toute librairie VHDL, les sources des paquetages sont fournies, avec des commentaires explicatifs, ce qui constitue une documentation d'utilisation. Les simulateurs travaillent généralement avec des procédures précompilées et optimisées pour accélérer leur exécution.

Contentons-nous ici d'expliquer le fonctionnement des trois procédures utilisées dans notre exemple<sup>1</sup> :

- `VitalSetupHoldCheck` analyse la disposition temporelle d'un signal par rapport à un front d'un signal de référence. Si des règles sont violées, cette procédure met à la valeur 'X' la variable de sortie. Le signal testé peut être un vecteur, sous réserve que tous les éléments du vecteur obéissent aux mêmes règles, ce qui est généralement le cas. Le front de référence est indiqué par le nom du signal concerné et un caractère conventionnel ('/' ici) qui indique la nature de la transition (front montant ordinaire, '0' → '1', ici).
- `VitalStateTable` est un moteur de machine d'états du type MOORE. La table de transitions fournit la valeur de l'état futur en fonction des entrées et de l'état actuel. Les différents niveaux possibles, et, pour les entrées, les fronts associés sont codés au moyen de caractères conventionnels définis dans le type `VitalTablesSymbolType`. Outre les niveaux classiques 'X', '0', '1' et '-' (*don't care*, ou indifférent) du type `std_logic`, on y trouve, par exemple : '/', pour un front montant binaire correct, '\', pour un front binaire descendant, 'S', pour l'absence de changement, '^' et 'r' pour des fronts montants entre niveaux métalogiques et 'F' pour un front descendant quelconque. La procédure analyse la table de haut en bas, et s'arrête à la première combinaison des entrées et de l'état actuel qui correspond à la situation. Cet ordre d'analyse permet de traiter les priorités et les commandes asynchrones. La variable `Result`, de mode `inout`, contient l'état de la machine : état initial avant l'appel puis état final après traitement de la transition.
- `VitalPathDelay` est une procédure très générale de mise à jour de la valeur d'un signal en fonction de retards et de chemins de causalités fournis par l'utilisateur. Elle permet de traiter des opérateurs à plusieurs entrées grâce à un tableau de chemins<sup>2</sup> qui contiennent chacun une date d'événement et des informations de retard associées. La procédure recherche le chemin qui crée en premier un événement en

1. Ces procédures ont plus d'arguments que ceux utilisés ici, les arguments inutilisés ont des valeurs par défaut qui étaient satisfaisantes pour notre propos. Le mécanisme de surcharge est employé dans la librairie pour pouvoir traiter des cas semblables mais avec des données différentes. Les machines d'états, par exemple, peuvent modéliser des situations bien plus complexes que celle présentée à propos d'une simple bascule : l'état de la machine peut être un vecteur, ce qui autorise, par exemple, la modélisation globale d'un compteur.
2. Un chemin décrit, indépendamment des équations logiques, les relations de cause à effet : à chaque sortie on attache la liste des entrées susceptibles d'en modifier la valeur, avec un certain retard.

sortie et actualise ce signal en conséquence. Elle mémorise les événements passés, ce qui lui permet de traiter les parasites éventuels (*glitches*) en fonction de directives qui lui sont passées dans des arguments optionnels (mode transport ou mode inertiel, par exemple).

Donnons, pour terminer cette description, un exemple de compte rendu de simulation obtenu avec la bascule précédente.

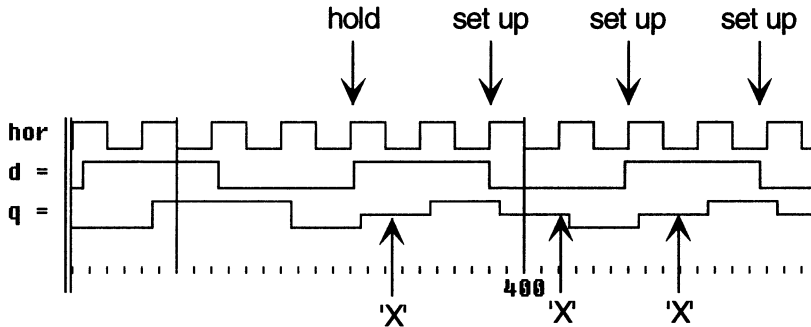


Figure 2-30 Violations des règles temporelles.

La figure 2-30 montre des exemples de traces obtenues; les messages associés précisent l'origine des fautes :

```
# ** Warning: :bascd: HOLD High VIOLATION ON d WITH RESPECT TO hor;
# Expected := 2 ns; Observed := 1 ns; At : 351 ns
# Time: 351 ns Iteration: 0 Instance:/
# ** Warning: :bascd: SETUP Low VIOLATION ON d WITH RESPECT TO hor;
# Expected := 8 ns; Observed := 0 ns; At : 390 ns
# Time: 390 ns Iteration: 0 Instance:/
# ** Warning: :bascd: SETUP High VIOLATION ON d WITH RESPECT TO hor;
# Expected := 8 ns; Observed := 1 ns; At : 430 ns
# Time: 430 ns Iteration: 0 Instance:/
# ** Warning: :bascd: SETUP Low VIOLATION ON d WITH RESPECT TO hor;
# Expected := 8 ns; Observed := 2 ns; At : 470 ns
# Time: 470 ns Iteration: 0 Instance:/
```

À des détails secondaires de présentation près, les trois versions de notre modèle de bascule D fournissent les mêmes résultats en simulation.

### ➤ Les fichiers de timings

Dans les modèles précédents, les paramètres temporels du circuit sont inclus, sous forme de paramètres génériques, dans le fichier source VHDL. Le standard VITAL offre la possibilité de fixer les valeurs de ces paramètres, sans avoir à modifier les programmes sources, au moyen de fichiers de spécifications temporelles. Ces fichiers obéissent au format SDF (*Standard Delay Format*), défini pour le langage VERILOG. La structure de ces fichiers repose sur un principe similaire à celui des configurations, dans une syntaxe complètement différente : pour chaque niveau

d'une hiérarchie, pour chaque composant, il est possible de préciser les valeurs de certains paramètres génériques.

Un simulateur compatible VITAL contient un interpréteur de fichiers SDF qui modifie les paramètres génériques des blocs en fonction des directives rencontrées. Le mécanisme suppose que l'utilisateur respecte des règles dans le choix des noms de ces paramètres : l'interpréteur attend des noms qu'il déduit automatiquement, par concaténation de chaînes de caractères, à partir de mots clés contenus dans les fichiers.

Rarement écrits à la main, ces fichiers de paramètres dynamiques sont d'une lisibilité moyenne, le langage n'étant pas très bavard. Le programme ci-dessous reprend l'exemple de la bascule D précédente en adoptant une convention de noms compatible avec un interpréteur VITAL :

```
library ieee,vital ;
use ieee.std_logic_1164.all ;
use ieee.VITAL_Timing.all;

entity bascd_sdf is
  generic (
    tpd_hor_q : VitalDelayType01 := (0 ns, 0 ns) ;
    tsetup_d_hor : time := 0 ns ; -- prépositionnement > 0
    thold_d_hor : time := 0 ns ) ; -- maintien
  port ( hor,d : in std_logic ;
        q : out std_logic ) ;
end bascd_sdf ;
```

Les seules modifications apportées sont les noms des paramètres génériques et la structure du premier qui est un vecteur d'éléments du type `time`, indexé par un type énuméré (`tr01`, `tr10`). Les deux éléments de ce vecteur contiennent les valeurs des deux temps de propagation `tplh` et `tphl`. Les noms des paramètres sont construits par des associations de mots clés (`tsetup`, `tpd` ...) et de noms des signaux impliqués dans la définition du paramètre temporel (`d` et `hor` dans notre exemple). Le code de l'architecture ne subit pas d'autres modifications que les adaptations, mineures, aux nouvelles définitions des paramètres, nous ne le reproduirons donc pas.

Les valeurs des génériques sont consignées dans un fichier SDF dont un exemple rudimentaire est donné ci-dessous :

```
(DELAYFILE
  (CELL(CELLTYPE "bascd_sdf")
    (INSTANCE *)
      (TIMINGCHECK
        (HOLD d hor (0:1:2) )
        (SETUP d hor (8:9:10) ))
        (DELAY(ABSOLUTE
          (IOPATH hor q (5:6:7) (7:8:9) ))
        )
      )
    )
  )
```

Dans chaque spécification de temps les valeurs sont regroupées par trois, qui correspondent aux options – minimum, typique et maximum – classiques dans les notices de circuits. Lors du chargement du modèle par le simulateur il est possible de choisir l'une de ces options pour chaque fichier de paramètre<sup>1</sup>.

### 2.7.7 Construire un « test bench »

La plupart des simulateurs logiques permettent de créer des stimuli pour vérifier le comportement d'un modèle. Souvent un langage de commande donne à l'utilisateur la possibilité d'automatiser, par la création de fichiers contenant des instructions de ce langage, l'enchaînement des opérations nécessaires au test.

Langage de modélisation, VHDL est également un langage de simulation. Il contient tous les éléments nécessaires à la création de stimuli, et, surtout, à l'exploitation des résultats. Nous en avons vu quelques exemples.

Dans ce qui suit nous n'apprendrons rien de nouveau concernant le langage, nous verrons comment utiliser les éléments connus pour construire des boîtes noires autonomes, qui en pilotent d'autres et en observent les réactions. Ces programmes de test sont traditionnellement nommés *test benches*.

#### a) Attention au mélange de genres

La première règle, que nous avons déjà évoquée, consiste à séparer nettement, dans une construction hiérarchique, les modules synthétisables de ceux qui servent à les essayer.

L'objectif que nous poursuivons ici est de construire un programme de test qui nous permette de contrôler le bon fonctionnement d'un module synthétisable. VHDL est un langage concurrent, ce point peut, au premier abord, déconcerter un habitué des langages de programmation séquentiels. Une méthode simple consiste à raisonner comme pour l'organisation d'un poste de mesure, dont les instruments réels seraient remplacés par des unités de conception, des programmes VHDL, interconnectés au sein d'une construction hiérarchique. Autrement dit, penser circuit, même virtuel, aide à penser la concurrence.

L'équipement minimum d'une installation destinée à vérifier le fonctionnement d'un circuit numérique comprend :

- Un générateur d'horloge, source d'impulsions périodiques dont on peut régler la fréquence.
- Un générateur de stimuli programmables. Piloté par l'horloge de cadencement général, ce générateur délivre des vecteurs de tests adaptés au fonctionnement du circuit à tester<sup>2</sup>.

1. Plusieurs fichiers peuvent être utilisés pour décrire des modules différents d'une hiérarchie.

2. Les concepteurs de circuits numériques savent que, VHDL ou pas, l'élaboration des vecteurs de tests est une tâche parfois aussi complexe et consommatrice de temps que la conception du circuit lui-même. Les placeurs routeurs contiennent souvent des outils d'aide à la conception de ces vecteurs.

- Un oscilloscope, ou, de préférence, un analyseur logique. Cet appareil de mesure multivoie permet d'explorer les signaux présents aux points accessibles du circuit. Le simulateur remplit cette fonction d'analyse de premier niveau. Outre son rôle de chef d'orchestre qui coordonne les fonctionnements des différents modules, il permet de placer des sondes (virtuelles) sur les équipotentielles du montage, de visualiser les résultats sous différentes formes (traces, listes de valeurs, ...), de placer des points d'arrêt, de définir des mots de déclenchement d'une capture, bref de faire tout ce qui est faisable avec un analyseur logique, l'accessibilité aux équipotentielles internes en plus<sup>1</sup>.
- Des appareils plus spécifiques de l'application concernée, analyseur de spectre par exemple.

Nous commencerons par constituer une boîte à outils rudimentaire que nous utiliserons pour contrôler le fonctionnement d'un opérateur particulier. Nous utiliserons ensuite les configurations pour rendre le banc de test réalisé indépendant des détails de fonctionnement de l'opérateur étudié. Une simple modification de la configuration permettra de valider le fonctionnement de circuits très différents, la seule contrainte étant qu'ils présentent une certaine régularité dans l'organisation de leurs ports d'accès. L'instrument d'analyse reste jusqu'à ce point le simulateur lui-même. Nous terminerons par un aperçu de tests plus « intelligents », qui fonctionnent en boucle fermée, modifiant, par exemple, la fréquence d'horloge jusqu'à trouver une différence entre le fonctionnement idéal d'un opérateur et son fonctionnement « réel », obtenu par un modèle rétroannoté.

### b) Une boîte à outils

Les deux premiers instruments nécessaires sont un générateur d'horloge et un générateur de séquences qui délivre les vecteurs de stimuli.

#### ► Les appareils

De façon à pouvoir régler la fréquence du signal délivré, le générateur d'horloge dont la description suit possède une entrée de type `time`, `tck`, qui fixe la période du signal de sortie `hor`. Si cette entrée est laissée déconnectée, sa valeur par défaut fixe la fréquence à 10 MHz :

```
library ieee ;
use ieee.std_logic_1164.all ;

entity genhor is
  port( tck : in time := 100 ns ;
        hor : buffer std_ulogic := '0') ;
end genhor ;

architecture simple of genhor is
begin
```

1. La réalité en moins, ces « mesures » restent virtuelles, ne l'oublions pas.

```

horloge : process
begin
  if tck > 0 ns then
    wait for tck/2 ; -- rapport cyclique fixé à 1/2
    hor <= not hor ;
  else
    wait on tck ;
  end if ;
end process horloge ;
end simple ;

```

Si la période est mise à zéro, ce qui serait absurde, le générateur s'arrête et attend une valeur non nulle pour redémarrer. Ce test permet de bloquer l'horloge de façon simple.

De façon à être versatile, le générateur de séquence est un peu plus compliqué. Il s'agit d'une machine de MOORE qui explore séquentiellement une table de vecteurs de tests (*testin*) qui lui est fournie en entrée. La dimension des vecteurs de test et leur nombre, qui fixe la longueur d'une séquence, sont fournis par des paramètres génériques, la définition du type de cette table est contenue dans le paquetage *tbenchpg*. À chaque front d'horloge ce générateur recopie en sortie (*vectout*) le vecteur courant, après un retard fixé par un paramètre générique. À tout moment le générateur peut être arrêté et/ou réinitialisé en début de séquence (entrées booléennes *stop* et *start*). Une sortie booléenne (*finseq*) indique la fin de la séquence en cours ; un simple rebouclage de cette sortie vers les entrées de commande permet d'obtenir une séquence unique ou un fonctionnement permanent.

```

library ieee ;
use ieee.std_logic_1164.all ;
use work.tbenchpg.all ;

entity genvect is
  generic(retard : time := 0 ns;
    taillevect : positive := 1;
    sequence : positive := 1 ) ;
  port (hor : in std_ulogic ;
    stop : in boolean := false ;
    start : in boolean := false ;
    testin : in vectable(1 to sequence, 1 to taillevect) ;
    vectout : out std_logic_vector(1 to taillevect) ;
    finseq : out boolean) ;
end genvect ;

architecture boucle of genvect is
  signal advect : positive := 1 ;
begin
  finseq <= advect = testin'high(1) ;
  stimuli : process
  begin
    for i in testin'range(2) loop
      vectout(i) <= testin(advect,i) after retard ;
    end loop
  end process
end architecture boucle ;

```



```

    end loop ;
    if not stop then
        if start then
            advect <= 1 ;
            elsif advect < testin'high(1) then
                advect <= advect + 1 ;
            end if ;
        end if ;
        wait until rising_edge(hor) ;
    end process stimuli ;
end boucle ;

```

Les déclarations utiles à la constitution d'un banc de test sont contenues dans le paquetage `tbenchpg` :

```

library ieee ;
use ieee.std_logic_1164.all ;

package tbenchpg is

    type vectable is array
        (positive range <>, positive range <>) of std_logic ;

    component genhor is
        port( tck : in time := 100 ns ;
              hor : buffer std_ulogic := '0') ;
    end component genhor ;

    component genvect is
        generic(retard : time := 0 ns;
              taillevect : positive := 1;
              sequence : positive := 1 ) ;
        port (hor : in std_ulogic ;
              stop : in boolean := false ;
              start : in boolean := false ;
              testin : in vectable(1 to sequence, 1 to taillevect) ;
              vectout : out std_logic_vector(1 to taillevect) ;
              finseq : out boolean) ;
    end component genvect ;

    component cobaye is
        port (hor : in std_ulogic ;
              commande : in std_logic_vector ;
              sorties : out std_logic_vector ) ;
    end component cobaye ;

    function tovectelem(e : in std_logic)
        return std_logic_vector ;

    function toelem(e : in std_logic_vector(1 to 1))
        return std_logic ;

```

```

end tbenchpg ;

package body tbenchpg is
  function tovectelem(e : in std_logic)
    return std_logic_vector is
    variable solo : std_logic_vector(1 to 1) ;
  begin
    solo(1) := e ;
    return solo ;
  end function tovectelem ;
  function toelem(e : in std_logic_vector(1 to 1))
    return std_logic is
    variable solo : std_logic ;
  begin
    solo := e(1) ;
    return solo ;
  end function toelem ;
end package body tbenchpg ;

```

Outre la définition du type de la table des vecteurs de test, ce paquetage contient les déclarations des composants qui interviennent dans le test et deux fonctions de conversion dont nous verrons l'utilité.

La déclaration du composant cobaye renseigne sur la structure des modules compatibles avec nos instruments : ils comportent une entrée d'horloge, un vecteur d'entrée et un vecteur de sortie de tailles quelconques. Les fonctions de conversion permettront également de tester des objets plus simples ne comportant qu'une entrée et/ou qu'une sortie binaires.

#### ➤ Un banc de test

La structure générale d'un banc de test, qui applique continuellement la même séquence à l'unité à tester, est celle de la figure 2-31.

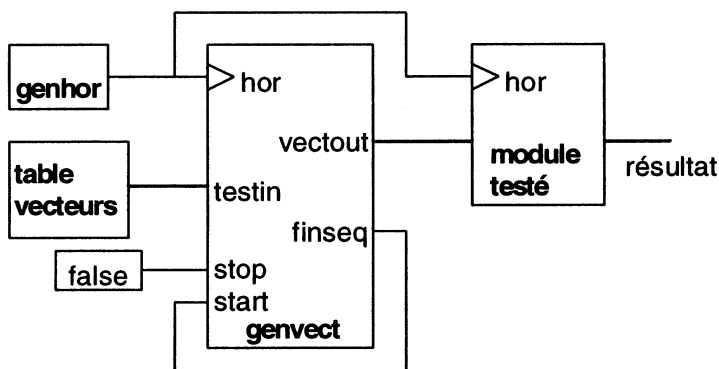


Figure 2-31 Banc de test.

Le programme ci-dessous utilise la structure de la figure 2-31 pour piloter un compteur binaire à sortie trois états et remise à zéro synchrone, dont le modèle a été donné à titre d'exemple d'introduction à la librairie IEEE (page 118) :

```

library ieee ;
use ieee.std_logic_1164.all,ieee.numeric_std.all ;
use work.tbenchpg.all ;

entity test_cpt is
end test_cpt ;

architecture struct of test_cpt is
    constant retard : time := 0 ns ;
    constant tck : time := 20 ns ;
    constant tailleresult : positive := 3 ;
    constant test : vectable := (
        -- fonction sortie
        "101", -- raz      active
        "001", -- mémoire active
        "011", -- comptage active
        "011", -- comptage active
        "011", -- comptage active
        "001", -- mémoire active
        "011", -- comptage active
        "010", -- comptage inactive
        "011", -- comptage active
        "111", -- raz      active
        "011"  -- comptage active
    ) ;

    constant taillevect : positive := test'high(2) ;
    constant sequence : positive := test'high(1) ;
    signal vectcom : std_logic_vector(1 to taillevect) ;
    signal testdata : vectable(1 to sequence,1 to taillevect)
        := test ;
    signal resultat : unsigned(tailleresult-1 downto 0) ;
    signal hor : std_ulogic ;
    signal stop : boolean := false ;
    signal roule : boolean ;

begin
    dut : entity
        work.compteur_tri(ieee_lib)
        generic map (taille => tailleresult)
        port map ( hor => hor,
            raz => vectcom(1),
            en => vectcom(2),
            oe => vectcom(3),
            compte => resultat
        ) ;

    horloge : genhor port map (tck => tck,
        hor => hor) ;

```

```

gene : genvect
  generic map (retard => retard,
               taillevect => taillevect,
               sequence => sequence)
  port map ( hor => hor,
             stop => stop,
             start => roule,
             testin => testdata,
             vectout => vectcom,
             finseq => roule) ;

end struct ;

```

On notera que toutes les dimensions utiles (signaux échangés, nombre de séquences) sont générées à la compilation à partir des définitions de la table et de quelques constantes.

La figure 2-32 montre le résultat d'une séquence de test qui utilise le programme précédent.

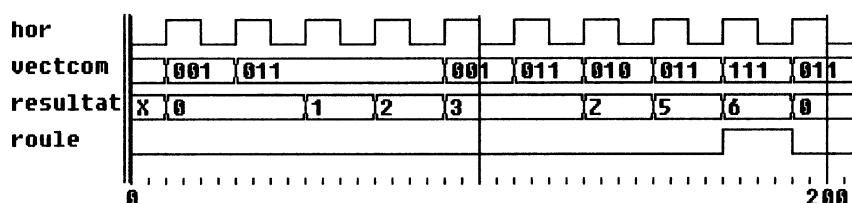


Figure 2-32 Test d'un compteur synchrone.

Des modifications simples permettent d'utiliser le même banc de test pour vérifier le fonctionnement d'une machine d'état comme le détecteur de trois zéros (page 81), dont le fonctionnement n'a pas grand rapport avec celui d'un compteur :

```

library ieee ;
use ieee.std_logic_1164.all,ieee.numeric_std.all ;
use work.tbenchpg.all ;

entity test_tz is
end test_tz ;

architecture struct of test_tz is
  constant tck : time := 20 ns ;
  constant retard : time := 0 ns ;
  constant tailleresult : positive := 1 ;
  constant test : vectable := (
    "1","0","0","0","1","0","0","0","0","0","0","0"
  ) ;
  constant taillevect : positive := test'high(2) ;
  constant sequence : positive := test'high(1) ;
  signal testdata : vectable(1 to sequence,1 to taillevect)
    := test ;

```



### c) Une configuration générale de test

Pour simplifier la gestion des interfaces entre le module à tester et son environnement, nous nous imposerons une structure homogène, qui est celle déclarée dans le paquetage `tbenchpg` sous le nom de `cobaye` : un composant accessible par une entrée d'horloge, un vecteur de commande et un vecteur de sortie. Pour faire rentrer le compteur dans ce cadre une simple modification de sa spécification d'entité fait l'affaire :

```
entity cnt_tri is
  generic(taille : natural := 10) ;
  port( hor : in std_ulogic ;
        cde : in std_logic_vector(1 to 3);
        compte : out unsigned(taille-1 downto 0) ) ;
  alias raz : std_logic is cde(1) ;
  alias en : std_logic is cde(2) ;
  alias oe : std_logic is cde(3) ;
end cnt_tri ;
```

La première étape consiste à créer un montage de test indépendant du module testé :

```
library ieee ;
use ieee.std_logic_1164.all,ieee.numeric_std.all ;
use work.tbenchpg.all ;

entity tbench is
  generic (retard : time := 0 ns ;
          tck : time := 20 ns ;
          tailleresult : positive := 2 ;
          test : vectable := ("00","01","10","11")) ;
  constant taillevect : positive := test'high(2) ;
  constant sequence : positive := test'high(1) ;
end tbench ;

architecture struct of tbench is
  signal vectcom : std_logic_vector(1 to taillevect) ;
  signal testdata :
    vectable(1 to sequence,1 to taillevect) := test ;
  signal resultat :
    std_logic_vector(tailleresult-1 downto 0) ;
  signal hor : std_ulogic ;
  signal stop : boolean := false ;
  signal roule : boolean ;
begin
  dut : cobaye
    port map ( hor => hor,
               commande => vectcom,
               sorties => resultat ) ;
  horloge : genhor port map (tck => tck,
                             hor => hor) ;
```

```

gene : genvect
  generic map (retard => retard,
               taillevect => taillevect,
               sequence => sequence)
  port map ( hor => hor,
             stop => stop,
             start => roule,
             testin => testdata,
             vectout => vectcom,
             finseq => roule) ;
end struct ;

```

La table des vecteurs de tests est définie par un paramètre générique, de façon à pouvoir être modifiée lors de l'utilisation de cette unité en tant que composant d'un banc de test. Dans la même logique, le composant à tester n'est relié à aucune unité de conception (*unbound*), une configuration permettra de créer les liens.

Un banc de test instancie le montage précédent comme un composant :

```

library ieee ;
use ieee.std_logic_1164.all ;
use work.tbenchpg.all ;

entity gen_test is end gen_test ;

architecture struct of gen_test is
  component benches is
  end component benches ;
begin
  banc_test : benches ;
end struct ;

```

Là encore, les liens entre composant et unité de conception sont laissés libres, le montage de test lui-même pourrait être modifié par une configuration.

Notre boîte à outils est complète. Une configuration permet de créer une séquence de test pour le compteur, en définissant les vecteurs à appliquer, la dimension du compteur, la fréquence d'horloge et le retard du générateur de séquences :

```

library ieee ;
use ieee.std_logic_1164.all, ieee.numeric_std.all ;
use work.tbenchpg.all ;

configuration cfgcttst of gen_test is
  for struct
    for banc_test : benches
      use entity work.tbench(struct)
      generic map (retard => 0 ns , -- réglages du banc
                   tck => 10 ns ,
                   tailleresult=> 3 ,
                   test => ("101","001","011","011",
                           "011","001","011","011",
                           "010","011","111","011")) ;
    end banc_test ;
  end struct ;
end configuration ;

```

```

    for struct
      for dut : cobaye use entity
        work.cnt_tri(ieee_lib) -- module testé
        generic map (taille => 3)
        port map ( hor => hor,
                  cde => commande,
                  std_logic_vector(compte) => sorties );
      end for ;
    end for ;
  end for ;
end configuration cfgcttst ;

```

Le détecteur de trois zéros est connecté à notre banc de mesures par une configuration différente :

```

library ieee ;
use ieee.std_logic_1164.all ;
use work.tbenchpg.all ;

configuration cfgtztst of gen_test is
  for struct
    for banc_test : benches
      use entity work.tbench(struct)
      generic map (retard => 0 ns , -- réglages du banc
                  tck => 10 ns ,
                  tailleresult=> 1 ,
                  test => ("1","0","0","0","0","0","0","1")) ;
    for struct
      for dut : cobaye use entity
        work.trois_zero(mealy3) -- module testé
        port map(hor => hor,
                  din => toelem(commande),
                  tovectelem(tzero) => sorties );
      end for ;
    end for ;
  end for ;
end configuration cfgtztst ;

```

Ce dernier module ne respecte pas le format général adopté pour les commandes et les sorties, qui sont dans son cas des scalaires. Les fonctions de conversions évoquées précédemment permettent de remédier à cette non-régularité<sup>1</sup>.

Une simulation de ces deux configurations fournit, bien évidemment, des résultats équivalents aux précédents, aux variantes de réglages du banc près.

1. Les programmes qui viennent d'être présentés nécessitent de la part du simulateur une grande souplesse, prévue par le standard VHDL, dans l'élaboration du schéma à simuler. Certains simulateurs acceptent mal le redimensionnement de tous les signaux d'un montage par un simple jeu de paramètres dans une configuration.



#### d) *Simulation en boucle fermée*

Les réglages du banc de « mesures » étaient jusqu'ici déterminés par des constantes et l'analyse des résultats laissée à la charge de l'utilisateur. Le langage permet également de créer des programmes de test qui effectuent un traitement de ces résultats, modifient dynamiquement les signaux de commande appliqués au module étudié, consignent ces résultats dans un fichier, etc.

Nous donnerons ci-dessous deux exemples simples de fonctionnement en boucle fermée. L'un consiste à rechercher automatiquement la fréquence maximum de fonctionnement d'un circuit, l'autre à simuler le comportement d'un microprocesseur pour contrôler le bon comportement d'un circuit périphérique qui génère des interruptions.

##### » Recherche d'une fréquence maximum de fonctionnement

L'exemple qui suit recherche la fréquence à partir de laquelle les sorties d'un circuit séquentiel diffèrent de leur modèle idéal. Tel quel le problème est, en fait, mal posé : tout circuit présente des retards, ce qui ne l'empêche pas nécessairement de remplir sa fonction. Il nous faut donc préciser un peu à partir de quand la différence, entre le comportement idéal et la réalité, devient inacceptable. Une règle simple est de considérer comme inadmissible une différence qui subsiste lorsque survient un front d'horloge. Deux situations radicalement différentes peuvent provoquer une telle situation :

- Les sorties sont des sorties directes des bascules qui matérialisent l'état interne du circuit. Dans ce cas une différence entre la réalité et le modèle provient d'un dysfonctionnement réel du circuit, l'état interne est faux. Les modèles rétroannotés testent, nous l'avons vu, les violations de *timing* des bascules. Ces modèles placent généralement leurs sorties à 'X' quand survient une telle violation, d'où une différence évidente. Cette situation est typique de ce que l'on observe pour un compteur placé dans un PLD simple ou modérément complexe.
- Entre les bascules et les sorties le schéma réel introduit des portes, soit pour réaliser une sortie décodée, soit tout simplement parce que le circuit dispose d'un routage entre les cellules logiques et les cellules d'entrée-sorties. Le retard entre un front d'horloge et les changements en sortie peut, dans ce cas, dépasser une période d'horloge, sans pour autant que la fonction réalisée soit fausse. Notre critère est, dans ce contexte, trop sommaire, il ne distingue pas les notions de fréquences maximums interne et externe, la seconde étant plus faible que la première. Cette situation s'observera pratiquement toujours dans les circuits complexes comme les FPGAs.

Malgré les réserves qui précèdent, nous conserverons l'idée simple de comparer le modèle et l'image de la réalité à chaque front d'horloge.

Le principe du test est résumé par la figure 2-34.

Le principe est simple, un comparateur observe à chaque front d'horloge les résultats attendu et réel ; quand il trouve une différence il positionne un indicateur booléen. Le système surveille cet indicateur et, quand survient une erreur, bloque l'horloge. Tant que tout va bien, la commande réduit de dix pour cent la fréquence d'horloge à chaque cycle.

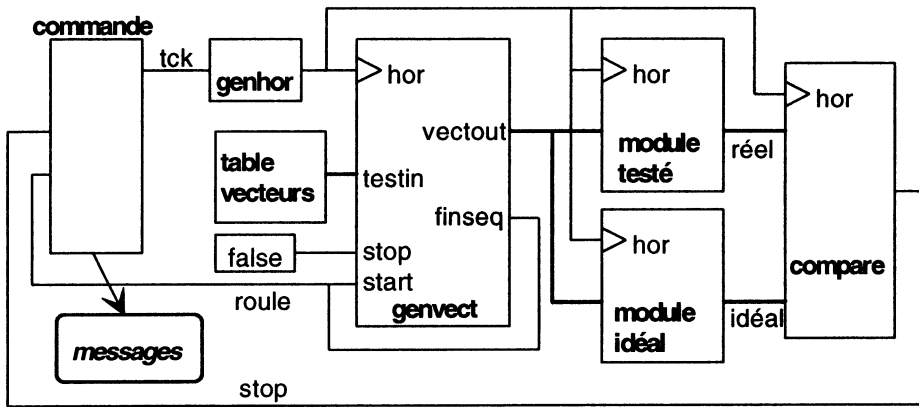


Figure 2-34 Banc de test en boucle fermée.

La boîte à outil doit être enrichie d'un module de comparaison :

```
library ieee ;
use ieee.std_logic_1164.all ;

entity compare is
    generic (tailleres : positive := 1) ;
    port ( hor : in std_ulogic ;
          ideal,reel : in std_logic_vector(1 to tailleres) ;
          erreur : out boolean := false ) ;
end compare ;

architecture simple of compare is
begin
    test : process
    begin
        wait until rising_edge(hor) ; -- un cycle à blanc
                                     -- pour l'initialisation

        loop
            wait until rising_edge(hor) ;
            erreur <= ideal /= reel ;
        end loop ;
    end process test ;
end simple ;
```

Le comparateur saute la première période d'horloge, pour laisser à l'utilisateur le loisir d'initialiser correctement les deux modules : beaucoup de modèles rétroannotés ne s'initialisent pas de la même façon que les modèles fonctionnels dont ils sont issus.

Le module testé est généré automatiquement par un placeur routeur (*fitter*), il s'agit soit d'un modèle « maison » soit, et c'est préférable, d'un modèle VITAL. Il est habituellement nécessaire d'adapter l'aspect extérieur (ports de l'entité) de façon

à reconstruire les noms d'origine<sup>1</sup>. Les placeurs routeurs ne conservent pas toujours les noms des signaux structurés, vecteurs tableaux et enregistrement sont parfois éclatés en leurs composantes binaires de type `std_logic`.

À titre d'illustration, le programme ci-dessous recherche la fréquence maximum de fonctionnement du compteur des exemples précédents :

```

library ieee ;
use ieee.std_logic_1164.all,ieee.numeric_std.all,
    std.textio.all ;
use work.tbenchpg.all ;
-- résolution du simulateur : une picoseconde
entity bench_bf is
end bench_bf ;

architecture struct of bench_bf is
    constant retard : time := 0 ns ;
    signal tck : time := 20 ns ;
    constant tailleresult : positive := 10 ;
    constant test : vectable := ("101","001","011","011",
                                "011","011","011","011",
                                "011","011","011","011"
                                ) ;
    constant taillevect : positive := test'high(2) ;
    constant sequence : positive := test'high(1) ;
    signal vectcom : std_logic_vector(1 to taillevect) ;
    signal testdata :
        vectable(1 to sequence,1 to taillevect) := test ;
    signal ideal, reel :
        std_logic_vector(tailleresult-1 downto 0) ;
    signal hor : std_ulogic ;
    signal stop : boolean := false ;
    signal roule : boolean ;

begin
    dut : entity
        work.CNT_TRIW(ARCH_CNT_TRIW) -- modèle rétroannoté
        port map ( hor => hor,
                    raz => vectcom(1),
                    en => vectcom(2),
                    oe => vectcom(3),
                    compte => reel
                    ) ;

    modele : entity
        work.compteur_tri(ieee_lib) -- modèle idéal
        generic map (taille => tailleresult)
        port map ( hor => hor,
                    raz => vectcom(1),
                    en => vectcom(2),

```

1. Nous avons vu, à propos du compteur, que les alias permettent de faire cela très simplement.

```

        oe => vectcom(3),
        std_logic_vector(compte) => ideal ) ;

verif : compare
    generic map(tailleres => tailleresult)
    port map (hor    => hor,
              ideal  => ideal,
              reel   => reel,
              erreur => stop ) ;

horloge : genhor port map (tck => tck,
                          hor => hor) ;

gene : genvect
    generic map (retard => retard,
                 taillevect => taillevect,
                 sequence => sequence)
    port map ( hor => hor,
              stop => false,
              start => roule,
              testin => testdata,
              vectout => vectcom,
              finseq => roule) ;

commande : process
    variable message : line ;
    variable periode : integer ;
begin
    wait until rising_edge(hor) ;
    if stop then
        periode := time'pos(tck) ; -- time -> entier
        write(message, 1e9/periode) ;
        report (" frequence maximum atteinte : " &
               message.all & " Mhz");
        deallocate (message) ;
        tck <= 0 ns ;
    elsif roule then
        tck <= tck - tck/10 ;
    end if ;
end process commande ;
end struct ;

```

Appliqué à un FPGA ce programme de test permet de trouver la fréquence maximum externe du compteur 10 bits :

```

# ** Note: frequence maximum atteinte : 76 Mhz
#   Time: 877531 ps Iteration: 0 Instance:/

```

Un détail du chronogramme de fonctionnement, représenté à la figure 2-35, montre la transition entre l'état 3 et l'état 4 du compteur, pour une période d'horloge de 14,6 ns (68 MHz).

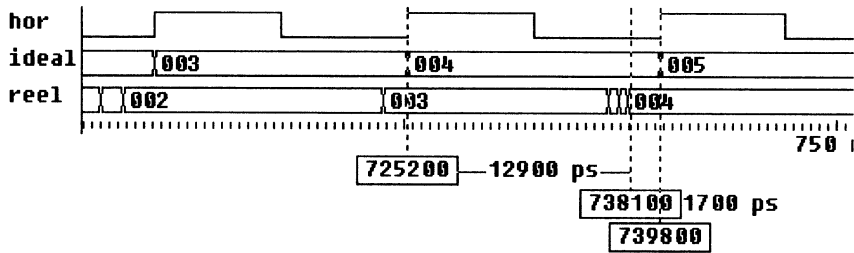


Figure 2-35 Modèle idéal et modèle rétroannoté, FPGA.

Cette figure met en évidence la différence entre retard des sorties et fonctionnement interne : le circuit utilisé dépasse largement les 120 MHz de fréquence de fonctionnement interne. Le retard de 13 ns correspond aux 76 MHz annoncé par le programme de test.

Le même compteur synthétisé dans un PLD 22V10 rapide cesse de fonctionner correctement avant que les sorties n'accusent un retard équivalent à la période d'horloge. La figure 2-36 montre les détails de la dernière transition sans erreur du compteur.

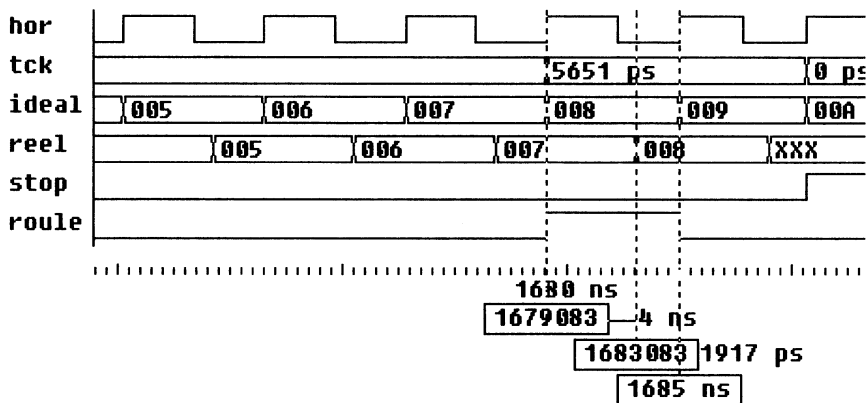


Figure 2-36 Modèle idéal et modèle rétroannoté, PLD.

À 176 MHz (une version très rapide du circuit, cela va sans dire...) le compteur se retrouve en situation de violation des règles temporelles alors que le retard des sorties est encore nettement inférieur à la période de l'horloge (4 ns contre 5,7 ns) :

```
# ** Warning: Setup ERROR ON D: Setting output to unknown
# Time: 1690691 ps Iteration: 0 Instance:/dut/mcell_23
# ** Note: frequence maximum atteinte : 176 Mhz
# Time: 1690691 ps Iteration: 1 Instance:/
```

Suite à ces erreurs, le modèle met les sorties à "X", ce qui provoque l'arrêt de l'horloge par le programme de test.

### ➤ Synchronisme et asynchronisme : transmetteur série

Dans un tout autre ordre d'idées, un programme de test en boucle fermée permet de simuler les réactions attendues de l'environnement d'un circuit en cours de conception, et par là de contrôler que le module réalisé se comporte correctement. Un exemple simple nous est fourni par un transmetteur série asynchrone, piloté par un microprocesseur. Le programme de test simule la réaction de l'unité centrale face à une demande d'interruption.

On souhaite réaliser la partie transmetteur, en version simplifiée, d'un circuit d'interface entre un microprocesseur et une liaison série RS-232.

Le circuit reçoit un signal d'horloge de fréquence égale à la cadence de transmission. Cette fréquence est très lente par rapport aux cycles d'accès bus du processeur, l'interface bus est donc obligatoirement asynchrone, le reste du fonctionnement est synchrone. Le synoptique général est représenté figure 2-37.

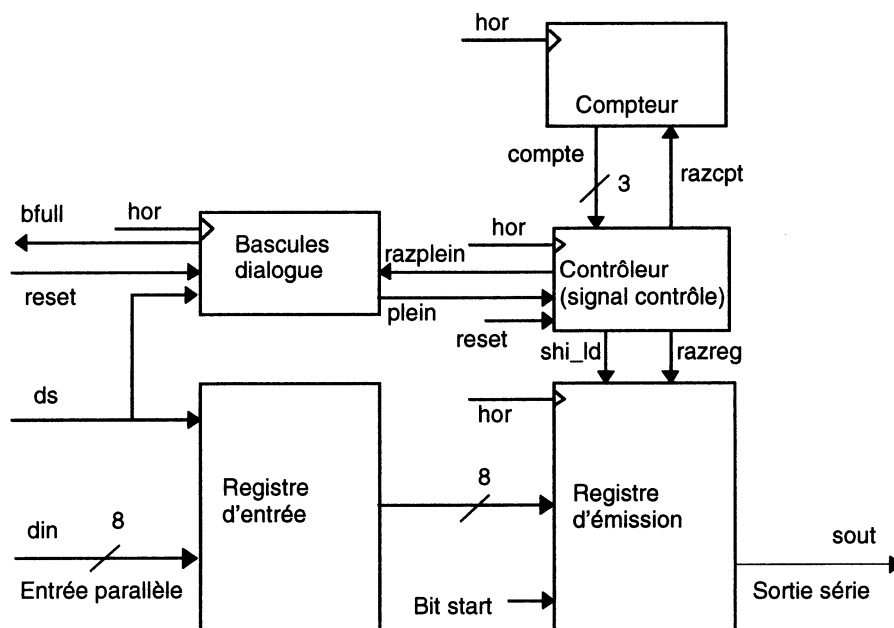
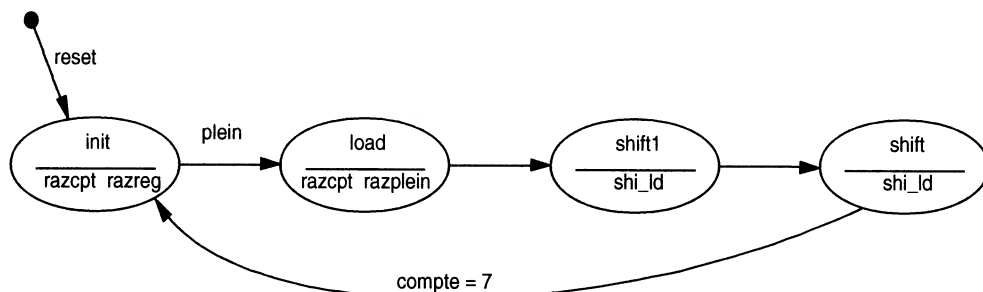


Figure 2-37 Transmetteur série asynchrone.

Le contrôleur est une machine d'états qui assure le chargement du registre d'émission, le décalage du signal de sortie et la libération du registre d'entrée quand son contenu a été transféré dans le registre d'émission. Le diagramme de transitions associé est celui de la figure 2-38.



**Figure 2-38** Transmetteur série, contrôleur.

En décrivant chaque bloc du schéma de principe par un processus, on peut représenter le fonctionnement du transmetteur par le code source (synthétisable) suivant :

```

entity ser_tx is
  port ( hor, ds, reset : in bit ;
        din : in bit_vector(7 downto 0) ;
        bfull: buffer bit ;
        sout : out bit ) ;
end ser_tx ;

architecture multi_proc of ser_tx is
  subtype etat is bit_vector(1 downto 0) ;
  constant init : etat := "11" ;
  constant load : etat := "01" ;
  constant shift1 : etat := "10" ;
  constant shift : etat := "00" ;
  signal controle : etat ; -- machine d'états contrôleur
  signal compte : integer
    range 0 to 7 ; -- 8 bits à émettre
  signal bufin : bit_vector(7 downto 0) ; -- latch entrée
  signal bufem : bit_vector(8 downto 0) ; -- reg. décalage
  signal razcpt : bit ; -- raz compteur émission
  signal shi_ld : bit ; -- commande décalage
  signal razreg : bit ; -- raz décalage
  signal razplein : bit ; -- raz bascule plein et bfull
  signal plein : bit ;
    -- bascule interne, synchrone buffer full
begin

  sout <= bufem(0) ;
  razcpt <= '1' when controle = init or controle = load
    else '0' ;
  razplein <= '1' when controle = load else '0' ;
  razreg <= '1' when controle = init else '0' ;
  shi_ld <= '0' when controle = load else '1' ;

```

```

buf_plein : process -- bascules dialogue partie synchrone
begin
    wait until hor = '1' ;
    if razplein = '1' then
        plein <= '0' ;
    elsif ds = '1' then
        plein <= bfull ;
    end if ;
end process buf_plein ;

rs_bfull : process ( ds, hor, reset)
begin -- bascules dialogue partie asynchrone
    if reset = '1' then
        bfull <= '0' ;
    elsif ds = '0' then
        bfull <= '1' ;
    elsif hor'event and hor = '1' then
        if razplein = '1' then
            bfull <= '0' ;
        end if ;
    end if ;
end process rs_bfull ;

buf_in : process (ds, din) -- registre entrée
begin
    if ds = '0' then -- chargement d'un octet
        bufin <= din ;
    end if ;
end process buf_in ;

compteur : process -- compteur bits émission
begin
    wait until hor = '1' ;
    if razcpt = '1' then
        compte <= 0 ;
    else
        compte <= (compte + 1) mod 8 ;
    end if ;
end process compteur ;

regem : process -- registre à décalage émission 9 bits
begin
    wait until hor = '1' ;
    if razreg = '1' then
        bufem <= (others => '0') ;
    elsif shi_ld = '0' then
        bufem(8 downto 1) <= bufin ; -- octet à émettre
        bufem(0) <= '1' ; -- bit start
    else
        bufem(8) <= '0' ;
    end if ;
end process regem ;

```



```

        for i in 0 to 7 loop
            bufem(i) <= bufem(i + 1) ;
        end loop ;
    end if ;
end process regem ;

controleur : process
begin
    wait until hor = '1' ;
    if reset = '1' then
        controle <= init ;
    else
        case controle is
            when init => if plein = '1' then
                            controle <= load ;
                        end if ;
            when load => controle <= shift1 ;
            when shift1 => controle <= shift ;
            when shift => if compte = 7 then
                            controle <= init ;
                        end if ;
        end case ;
    end if ;
end process controleur ;
end multi_proc ;

```

Pour tester le fonctionnement du circuit précédent on fait appel à un *test bench* qui simule l'horloge et le microprocesseur. Après une réinitialisation du circuit, le processeur simulé charge successivement trois octets à émettre, en laissant suffisamment de temps pour que l'émission se fasse sans problème. Dans une deuxième phase le processeur émet un nouvel octet sur demande du circuit (signal occupe à zéro), simulant par là un fonctionnement par interruption.

```

entity test_tx is
end test_tx ;

architecture micro of test_tx is
    signal datas : bit_vector (7 downto 0) ; -- donnees à émettre
    signal hor, serie, occupe, ds, reset : bit ;
    component ser_tx is
        port ( hor, ds, reset : in bit ;
              din : in bit_vector(7 downto 0) ;
              bfull: buffer bit ;
              sout : out bit ) ;
    end component ser_tx ;
begin
    reset <= '1', '0' after 20 us ; -- initialisation

    emetteur : ser_tx port map
        (hor => hor , ds => ds, reset => reset,
         bfull => occupe ,
         sout => serie ,
         din => datas ) ;

```

```

horloge : process -- générateur de période 10 us
begin
    wait for 5 us ;
    hor <= not hor ;
end process horloge ;

proc : process
begin
    datas <= X"C3" after 0 ns ;
    ds <= '1' ;      -- fonctionnement en boucle ouverte
    wait for 30 us ; -- attente
    ds <= '0' ;      -- écriture
    wait for 100 ns ; -- fonctionnement en boucle ouverte
    ds <= '1' ;
    datas <= not datas after 10 us ;
    wait for 100 us ; -- attente
    ds <= '0' ;      -- écriture
    wait for 100 ns ;
    ds <= '1' ;
    datas <= not datas after 10 us ;
    wait for 200 us ; -- attente
    ds <= '0' ;      -- écriture
    wait for 100 ns ;
    ds <= '1' ;
    datas <= not datas after 10 us ;
    loop          -- fonctionnement en boucle fermée
        wait until occupe = '0' ;
        ds <= '0' after 1 us, '1' after 1100 ns ;
        datas <= not datas after 10 us ;
    end loop ;
end process proc ;
end micro ;

```

Les chronogrammes de test, figure 2-39, mettent en évidence les vitesses très différentes de fonctionnement du processeur (rapide) et de l'émetteur (lent). Cette différence de vitesses nécessite l'emploi d'organes de dialogue peu orthodoxes, qui mélangent des fonctionnements synchrone et asynchrone.

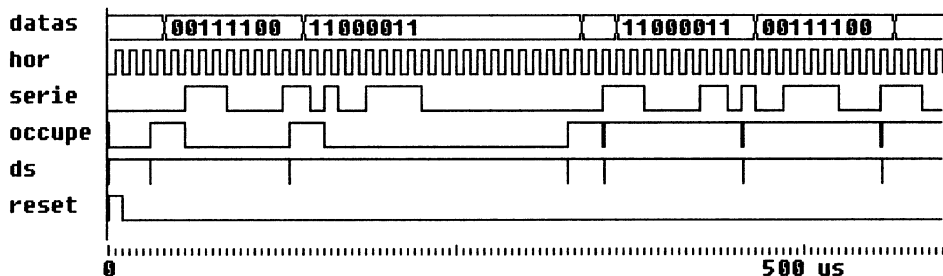


Figure 2-39 Transmetteur série, séquence de test.

Paradoxalement la partie la plus délicate du programme se situe dans l'écriture du code de ces deux bascules de dialogue. La recherche d'erreurs de principe est grandement facilitée par la possibilité de créer des tests en boucle fermée.

### c) En matière de conclusion

Du langage au circuit, du circuit au langage. VHDL est riche, trop diront certains, peut-être. L'utilisateur est aidé, et le sera de plus en plus, par des générateurs automatiques de code. Il en existe de nombreux qui traduisent des schémas blocs en programmes structurels, des diagrammes de transitions en programmes comportementaux et, bien sûr, des *net-list* en modèles rétroannotés. Il en est de même en programmation. Comme là, l'utilisateur doit être capable de « rentrer dans le code », d'en comprendre le sens, de savoir reprendre la main et d'apporter ses modifications.

Notre expérience de formations à des niveaux variés nous ont convaincu qu'une approche en douceur était possible. À une condition : aller du simple au compliqué, du synthétisable à la modélisation. Les difficultés rencontrées, tout arrive, proviennent plus souvent d'une mécompréhension du fonctionnement des circuits électroniques que du langage. On peut tout décrire en VHDL, des circuits sains comme des sources inépuisables d'aléas, rappelons ces règles de base :

- Diviser pour régner, seuls les modules petits sont testables et réparables.
- Privilégier les solutions synchrones, seuls certains opérateurs d'interfaces nécessitent des fonctionnements asynchrones.
- Utiliser des programmes de test en cherchant les fautes, un programme doit *a priori* être considéré comme défaillant, reste à trouver où. Notre penchant naturel nous pousse plutôt à présumer l'inverse, qu'en l'absence de preuve contraire notre œuvre est correcte.
- Les compilateurs de synthèse sont moins rigoureux que les bons simulateurs. Une bonne pratique est de tester systématiquement les programmes avec un simulateur, en veillant à limiter les instructions des modules qui décrivent de futurs circuits au sous-ensemble accepté en synthèse.

## Chapitre 3

---

# Les circuits programmables

L'histoire des circuits programmables commence, à peu de choses près, avec la décennie quatre-vingt. À la fin des années soixante-dix le monde des circuits numériques se répartit schématiquement en quatre grands groupes :

- Les fonctions standard sont utilisées pour les applications réalisées en logique câblée. Les catalogues TTL et CMOS présentent plusieurs centaines de fonctions d'usage général, sous forme de circuits intégrés à grande échelle.
- Les microprocesseurs, désormais d'usage courant, et sont omniprésents dans les applications industrielles.
- Dans des applications trop complexes pour être raisonnablement traitées en logique câblée traditionnelle, et trop rapides pour avoir une solution à base de microprocesseurs, on utilise des séquenceurs microprogrammés.
- Quand les volumes de production importants le justifient, les circuits intégrés spécifiques (ASICs) offrent une alternative aux cartes câblées classiques.

Le développement des microprocesseurs stimule une évolution rapide des technologies de réalisation des mémoires à semi-conducteurs ; les circuits logiques programmables ont hérité directement des mémoires pour ce qui concerne les aspects technologiques. Leurs architectures internes sont, en revanche, très différentes. Il n'est donc pas surprenant que le premier fabricant de circuits programmables ait été un fabricant de mémoires (MMI, *monolithic memories inc.*). Notons, pour la petite histoire, que cette société a « fusionné<sup>1</sup> » avec l'un des leaders des fabricants de processeurs rapides, processeurs microprogrammés, processeurs

---

1. Fusion entre un géant et un nain.

spécialisés dans le traitement de signal, processeurs RISCs, AMD (*advanced micro devices*).

Ce chapitre est consacrée à un tour d'horizon général au cours duquel nous tenterons de donner au lecteur une vision d'ensemble de ce que sont les membres de la famille (nombreuse) des circuits programmables.

Les aspects structurels internes ne seront abordés que dans la mesure où ils éclaireront la compréhension du fonctionnement. L'utilisateur d'un circuit a principalement besoin de bien dominer son architecture, les technologies de fabrication appartiennent au domaine du fondeur.

### 3.1 LES GRANDES FAMILLES

Indépendamment de sa structure interne et des détails de la technologie concernée, une mémoire est caractérisée par son mode de programmation et sa faculté de retenir l'information quand l'alimentation est interrompue. Les catégories de mémoires qui ont donné naissance aux circuits programmables sont :

- Les mémoires de type PROM (*programmable memory*) sont programmables une seule fois au moyen d'un appareil spécial, le programmeur<sup>1</sup>. Les données qui y sont inscrites ne sont pas modifiables. Elles conservent les informations quand l'alimentation est interrompue.
- Leur inconvénient majeur est l'impossibilité de modifier les informations qu'elles contiennent.
- Les mémoires de type EPROM (*erasable programmable memory*) sont programmables par l'utilisateur au moyen d'un programmeur, effaçables par une exposition aux rayons ultraviolets et reprogrammables après avoir été effacées. Elles aussi conservent les informations quand l'alimentation est interrompue.
- Leur boîtier doit être équipé d'une fenêtre transparente, ce qui en augmente le coût. La modification de leur contenu est une opération longue qui nécessite des manipulations : plusieurs minutes pour l'effacement des données anciennes, sur un premier appareil, et transfert de nouvelles informations sur un second appareil.
- Les mémoires de type EEPROM (*electrically erasable programmable memory*), ou FLASH, sont effaçables et reprogrammables électriquement. Non alimentées, elles conservent les informations mémorisées.
- La diminution des tensions à appliquer pour programmer les mémoires FLASH permet même de s'affranchir du programmeur : il est intégré dans le circuit. On parle alors de mémoires programmables *in situ* (ISP, pour *in situ programming*), c'est-à-dire sans démonter la mémoire de la carte sur laquelle elle est implantée.

---

1. Quelle que soit la technologie, la programmation d'un circuit consiste à lui faire subir des niveaux de tension et de courant qui sortent du cadre de son utilisation normale. Typiquement, les tensions mises en jeu vont de 10 (CMOS) à 20 volts (bipolaires), les courants correspondants de quelques dizaines à quelques centaines de milliampères, pour des circuits alimentés sous 5 volts en fonctionnement ordinaire.

- Même programmable *in situ*, une mémoire FLASH fonctionne dans un mode tout à fait différent du mode « utilisation » quand elle est en cours de programmation. Les technologies FLASH sont de loin les plus séduisantes pour les circuits programmables pas trop complexes.
- Les mémoires RAM (*random access memory*) statiques<sup>1</sup>, ou SRAM, sont constituées de cellules accessibles, en mode normal, en lecture et en écriture. Elles sont utilisées dans certains circuits programmables complexes pour conserver la configuration (qui définit la fonction réalisée) du circuit.

Ces mémoires perdent leur information quand l'alimentation est supprimée. Les circuits qui les utilisent doivent donc suivre un cycle d'initialisation à chaque mise sous tension. Ces circuits peuvent être reconfigurés dynamiquement, changeant ainsi de fonction à la demande, en cours d'utilisation.

Comme tout domaine spécialisé, le monde des circuits programmables comporte une terminologie, d'origine anglo-saxonne le plus souvent, difficile à éviter. Pour compliquer les choses, de nombreux termes sont à l'origine des noms propres, propriétés des sociétés qui sont à la source des produits concernés. Qui se souvient encore que frigidaire est un nom déposé par la société General Motors ?

Les sigles utilisés dans la suite semblent communément admis par la majorité des fabricants :

- PLD (*programmable logic device*) est un terme générique qui recouvre l'ensemble des circuits logiques programmables. Il est le plus souvent employé pour désigner les plus simples d'entre eux (équivalent de quelques centaines de portes logiques).
- CPLD (*complex programmable logic device*) désigne évidemment un circuit relativement complexe (jusqu'à une ou deux dizaines de milliers de portes), mais dont l'architecture dérive directement de celle des PLDs simples.
- FPGA (*field programmable gate array*) marque un saut dans l'architecture et la technologie, il désigne un circuit qui peut être très complexe (jusqu'à cent mille portes équivalentes) ; la complexité des FPGAs rejoint celle des ASICs (*application specific integrated circuits*).

Notons que la complexité d'un circuit n'est pas mesurable simplement : il ne suffit pas qu'un circuit contienne un grand nombre de portes pour être puissant ; encore faut-il que ces portes soient utilisables dans une grande proportion. Ce dernier point est à la fois un problème d'architecture et de logiciels d'aide à la conception.

---

1. L'adjectif statique s'oppose à dynamique. Les mémoires dynamiques stockent les informations dans des capacités - intrinsèquement liées aux structures MOS - qui nécessitent un processus dynamique de rafraîchissement. Les cellules des mémoires statiques sont des bistables, qui conservent leur état tant que l'alimentation est présente.

## 3.2 QU'EST-CE QU'UN CIRCUIT PROGRAMMABLE ?

Un circuit programmable est un assemblage d'opérateurs logiques combinatoires et de bascules dans lequel la fonction réalisée n'est pas fixée lors de la fabrication. Il contient potentiellement la possibilité de réaliser toute une classe de fonctions, plus ou moins large suivant son architecture. La programmation du circuit consiste à définir une fonction parmi toutes celles qui sont potentiellement réalisables.

Comme dans toute réalisation en logique câblée, une fonction logique est définie par les interconnexions entre des opérateurs combinatoires et des bascules (synchrones, cela va presque sans dire), et par les équations des opérateurs combinatoires. Ce qui est programmable dans un circuit concerne donc les interconnexions et les opérateurs combinatoires. Les bascules sont le plus souvent de simples bascules D, ou des bascules configurables en bascules D ou T.

La réalisation d'opérateurs combinatoires utilise des opérateurs génériques, c'est à eux que nous allons nous intéresser dans la suite.

### 3.2.1 Des opérateurs génériques

Les opérateurs combinatoires génériques qui interviennent dans les circuits programmables proviennent soit des mémoires (réseaux logiques) soit des fonctions standards (multiplexeurs et ou exclusif).

#### a) Réseaux logiques programmables

Un réseau logique programmable (PLA, pour *programmable logic array*<sup>1</sup>) utilise le fait que toute fonction logique combinatoire peut se mettre sous forme d'une somme (OU logique) de produits (ET logique), c'est ce que l'on appelle classiquement la première forme normale, ou forme disjonctive.

Le schéma de la figure 3-1 représente une structure de PLA simple. La programmation du circuit consiste à supprimer certaines des connexions marquées d'une croix. Si une connexion est supprimée, une valeur constante '1' est appliquée à l'entrée correspondante de la porte ET, c'est ce que symbolise le réseau de résistances relié à cette valeur constante.

Un tel schéma permet de réaliser n'importe quelle fonction booléenne  $s(e1, e2)$ , de deux variables binaires  $e1$  et  $e2$ <sup>2</sup>, pourvu qu'elle ne dépasse pas deux termes.

---

1. Que l'on ne confondra pas avec PAL (*programmable array logic*), qui désigne les PLDs historiques de MMI...Ni avec GAL (*gate array logic*) nom déposé par la société Lattice, etc.

2. Dans tout cet ouvrage, sauf précision contraire, nous utiliserons les valeurs 0 et 1 pour représenter les états possibles d'une variable binaire. Les opérateurs ET et OU sont définis avec la convention  $0 \leftrightarrow \text{FAUX}$  et  $1 \leftrightarrow \text{VRAI}$ . Les circuits associent bien sûr les valeurs logiques à des niveaux électriques; sauf précision contraire, nous prendrons une convention logique positive qui associe 1 à un niveau haut (H pour *high*) et 0 à un niveau bas (L pour *low*).

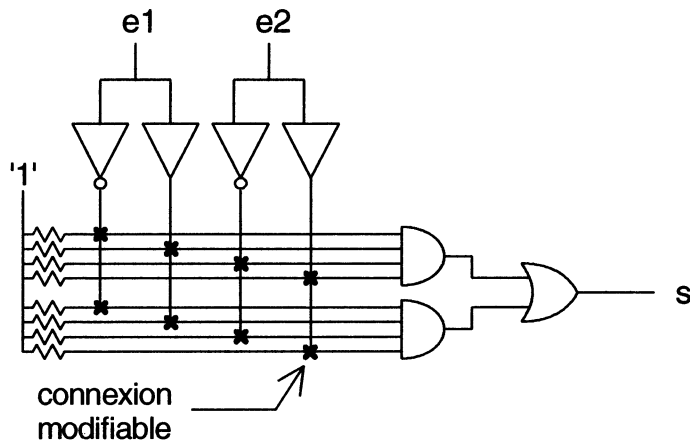


Figure 3-1 Un PLA simple.

En effet, si toutes les connexions sont présentes, en notant par + et \* les opérateurs OU et ET, respectivement, s s'écrit :

$$s(e1, e2) = \overline{e1} * e1 * \overline{e2} * e2 + \overline{e1} * e1 * \overline{e2} * e2$$

qui vaut trivialement '0'.

Un opérateur ou exclusif, par exemple, obéit à l'équation :

$$e1 \oplus e2 = \overline{e1} * e2 + e1 * \overline{e2}$$

d'où la programmation du PLA de la figure 3-2.

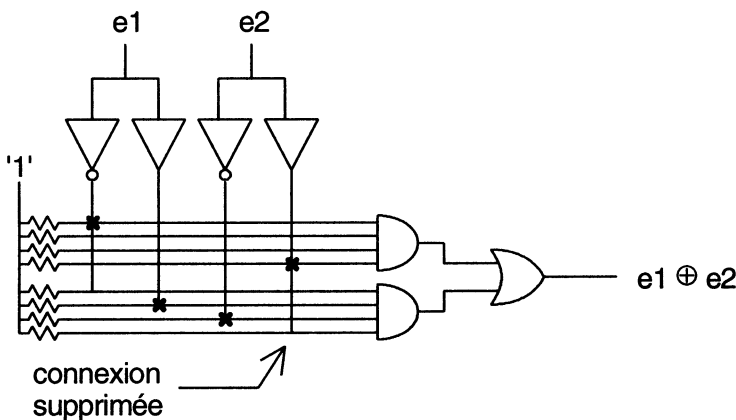


Figure 3-2 PLA réalisant un ou exclusif.

Le plus simple des PLDs, un 16L8 par exemple, utilise des opérateurs ET à 32 entrées et des opérateurs OU à 8 entrées. Un schéma tel que celui des figures précédentes deviendrait, dans de telles conditions, illisible. Pour éviter cet écueil, les



notices de circuits utilisent des symboles simplifiés, pour représenter les réseaux logiques programmables.

La figure 3-3<sup>1</sup> représente le PLA précédent avec ces symboles.

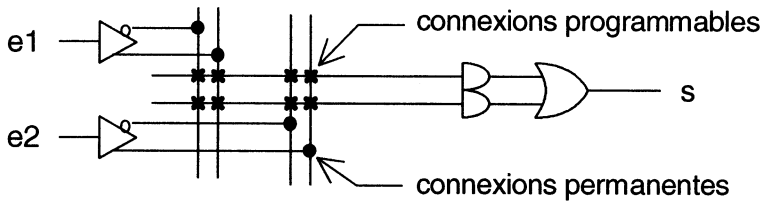


Figure 3-3 Symbole d'un PLA 2 × 4.

Dans un tel schéma, toutes les entrées (et leurs compléments) peuvent être connectées à tous les opérateurs ET par programmation. Par référence à la première technologie utilisée, ces connexions programmables portent le nom de fusibles, même quand leur réalisation n'en comporte aucun. Quand il s'agit uniquement d'illustrer la structure d'un circuit programmable, et non la réalisation d'une fonction particulière, les croix qui symbolisent les fusibles ne sont même pas représentées.

Dans cette évocation simplifiée, le schéma de l'opérateur ou exclusif devient celui de la figure 3-4, dans laquelle une croix représente une connexion programmable maintenue, l'absence de croix une connexion supprimée.

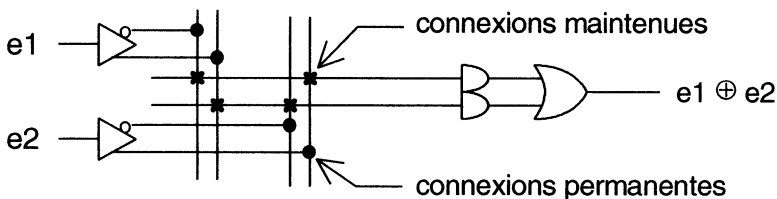


Figure 3-4 PLA 2 × 4 réalisant un ou exclusif.

### b) Multiplexeurs

Un multiplexeur est un aiguillage d'informations. Dans sa forme la plus simple, il comporte deux entrées de données, une sortie et une entrée de sélection, conformément au symbole de la figure 3-5.

Le fonctionnement de cet opérateur se décrit très simplement sous une forme algorithmique :

si sel = '0'	$s \leftarrow in0;$
si non (sel = '1')	$s \leftarrow in1;$

1. Le nombre 2 × 4 indique la dimension de la matrice de fusibles : deux lignes et quatre colonnes.

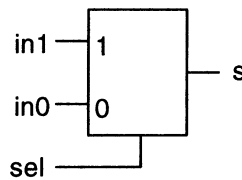


Figure 3-5 Un multiplexeur élémentaire.

Certains constructeurs notent sur le symbole, comme nous l'avons fait, la valeur de l'entrée de sélection en regard de l'entrée correspondante.

La première utilisation des multiplexeurs dans les circuits programmables est, évidemment, de créer des chemins de données. La programmation consiste alors à fixer des valeurs aux entrées de sélection.

Une autre utilisation de la même fonction consiste à remarquer qu'un multiplexeur est, en soi, un opérateur générique. Reprenant l'exemple précédent du *ou exclusif*, on peut le décrire sous forme algorithmique :

si $e1 = '0'$	$e1 \oplus e2 \leftarrow e2$ ;
si non ( $e1 = '1'$ )	$e1 \oplus e2 \leftarrow \overline{e2}$ ;

D'où une réalisation possible de l'opérateur *ou exclusif* au moyen d'un multiplexeur dans le schéma de la figure 3-6.

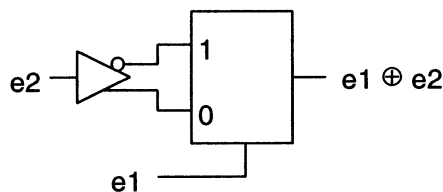


Figure 3-6 Ou exclusif réalisé par un multiplexeur.

L'exemple précédent peut être généralisé sans peine : un multiplexeur à  $n$  entrées de sélection, soit  $2^n$  entrées de données, permet de réaliser n'importe quelle fonction combinatoire de  $n + 1$  entrées, pourvu que l'une, au moins, de ces entrées existe sous forme directe et sous forme complémentée.

Dans un circuit programmable dont les « briques » de base sont des multiplexeurs (c'est le cas de beaucoup de FPGAs) la programmation consiste à fixer des chemins de données, c'est-à-dire à établir des interconnexions entre des cellules de calcul et des signaux d'entrée et de sortie. Cette opération de création d'interconnexions entre des cellules génériques s'appelle le *roulage* d'un circuit ; l'affectation des cellules à des fonctions souhaitées par l'utilisateur s'appelle le *placement*.

## c) Ou exclusif

L'opérateur élémentaire *ou exclusif*, ou somme modulo 2, dont nous avons rappelé l'expression algébrique précédemment, est disponible en tant que tel dans certains circuits.

Cet opérateur intervient naturellement dans de nombreuses fonctions combinatoires reliées de près ou de loin à l'arithmétique : additions et soustractions, contrôles d'erreurs, cryptages en tout genre, etc. Or ces fonctions se prêtent mal à une représentation en somme de produits, car elles ne conduisent à aucune minimisation de leurs équations<sup>1</sup>. Un simple générateur de parité sur 8 bits, qui rajoute un bit de parité à un octet de données, ne nécessite pas moins de 128 ( $2^{8-1}$ ) produits, quand il est « mis à plat », pour être réalisé au moyen d'une couche de ETs et d'une couche de OUs. De nombreuses familles de circuits programmables disposent, en plus des PLAs, d'opérateurs *ou exclusifs* pour faciliter la réalisation de ces fonctions arithmétiques.

Une autre application de l'opérateur *ou exclusif* est la programmation de la polarité d'une expression. Quand on calcule une fonction combinatoire quelconque sous forme disjonctive, il peut arriver qu'il soit plus économique, en nombre de produits nécessaires, de calculer le complément de la fonction et de complémenter le résultat obtenu.

Par exemple, si  $(cba)_2$  représente l'écriture en base 2 d'un nombre N, compris entre 0 et 7, on peut exprimer par une équation logique que N est premier avec 3 :

$$\text{prem3} = c * \bar{b} + \bar{b} * a + c * a + \bar{c} * b * \bar{a}$$

On peut également remarquer que si N est premier avec 3 c'est qu'il n'est pas multiple de 3, soit :

$$\text{prem3} = \text{mul3} = \overline{c * \bar{b} * \bar{a} + \bar{c} * b * a + c * b * a}$$

La deuxième forme, apparemment plus complexe, nécessite un produit de moins que la première pour sa réalisation. Dans des circuits où le nombre de produits disponibles est limité, cela peut présenter un avantage.

Un opérateur *ou exclusif* permet de passer, par programmation, d'une expression à son complément, comme l'indique la figure 3-7. Comme cet opérateur peut être réalisé avec un multiplexeur, l'une ou l'autre de ces formes peut se trouver dans les notices !

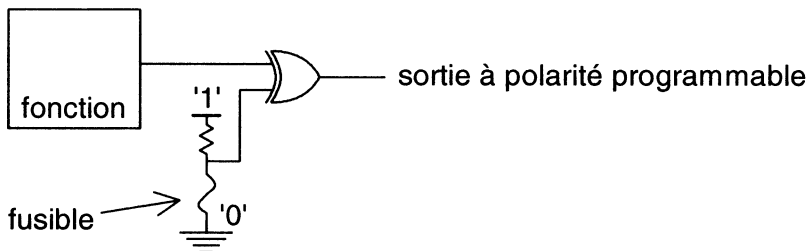


Figure 3-7 Polarité programmée par un ou exclusif.

1. Au moyen de tableaux de Karnaugh, ou, de façon plus réaliste, par des programmes de minimisation qui utilisent des algorithmes comme celui de Queens et Mc Cluskey ou ESPRESSO.

#### d) Bascules

Qui dit logique dit logique séquentielle. Les circuits programmables actuels offrent tous la possibilité de créer des fonctions séquentielles, synchrones dans leur immense majorité. La brique de base de toute fonction séquentielle est la bascule, cellule mémoire élémentaire susceptible de changer d'état quand survient un front actif de son signal d'horloge.

Bascule D, T ou J-K ? La première est toujours présente. Comme certaines fonctions se réalisent plus simplement avec la seconde, les compteurs par exemple, de nombreux circuits permettent, toujours par programmation, de choisir entre bascule D et bascule T<sup>1</sup>, voire entre l'un des trois types de base. La figure 3-8 rappelle, par un diagramme de transitions, le fonctionnement de ces trois types de bascules.

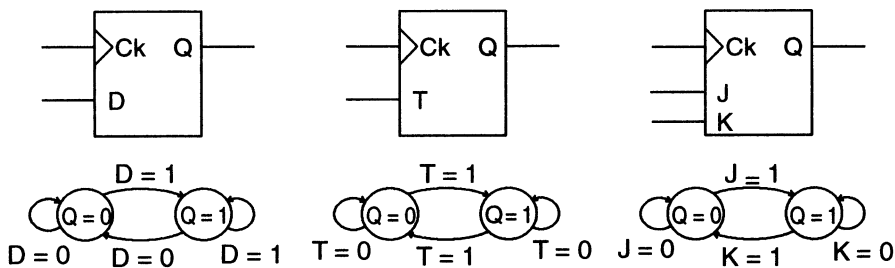


Figure 3-8 Les trois types fondamentaux de bascules.

Le programmeur n'a, en réalité, que rarement à se préoccuper de ce genre de choix, les optimiseurs déterminent automatiquement le type de bascule le mieux adapté à l'application.

### 3.2.2 Des technologies

Premier critère de choix d'un circuit programmable, la technologie utilisée pour matérialiser les interconnexions détermine les aspects électriques de la programmation : maintien (ou non) de la fonction programmée en l'absence d'alimentation, possibilité (ou non) de modifier la fonction programmée, nécessité (ou non) d'utiliser un appareil spécial (un programmeur, bien sûr).

#### a) Fusibles

Première méthode employée, la connexion par fusibles, est en voie de disparition. On ne la rencontre plus que dans quelques circuits de faible densité, de conception ancienne.

1. C'est un (bon) exercice de logique séquentielle élémentaire que de trouver le schéma de n'importe quel type de bascule en utilisant n'importe quel autre type.

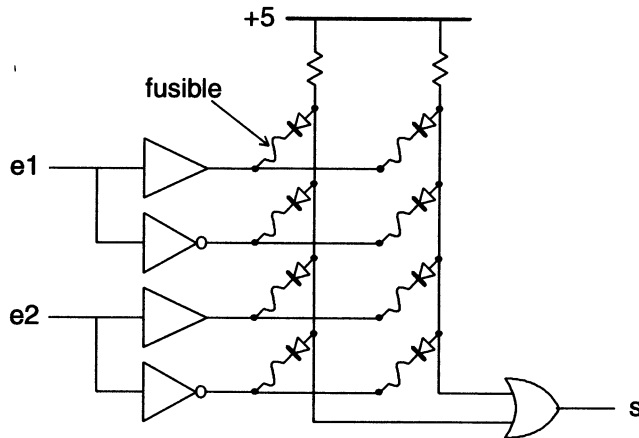


Figure 3-9 Pld élémentaire à fusibles.

La figure 3-9 en illustre le principe ; toutes les connexions sont établies à la fabrication. Lors de la programmation le circuit est placé dans un mode particulier par le programmeur<sup>1</sup>, mode dans lequel des impulsions de courant sont aiguillées successivement vers les fusibles à détruire.

Pour programmer un circuit, il faut transférer dans le programmeur une table qui indique par un chiffre binaire l'état de chaque fusible : la table des fusibles. Cette table est généralement transférée entre le système de CAO et le programmeur sous forme d'un fichier au format normalisé : le format JEDEC.

L'exemple ci-dessous est un extrait du fichier JEDEC, généré par un compilateur VHDL, qui implémente un compteur binaire 10 bits dans un circuit de type 22V10 :

```

_C22V10*
QP24*           Number of Pins*
QF5828*         Number of Fuses*
F0*             Note: Default fuse setting 0*
G0*             Note: Security bit Unprogrammed*
NOTE DEVICE C22V10*
NOTE PACKAGE PAL22V10G-5PC*
NOTE PINS hor:1 oe:2 en:3 raz:4 compte_6:14 compte_8:15 compte_9:16
compte_3:17 *
NOTE PINS compte_1:18 compte_0:19 compte_2:20 compte_4:21 compte_7:22 *
NOTE PINS compte_5:23 *
NOTE NODES *
L00000

```

1. Ce mode est activé par une tension supérieure à la normale, appliquée sur une broche particulière du circuit. Dans ce mode, les autres broches servent à fournir au circuit le numéro du fusible à détruire et à appliquer une impulsion qui provoque une surintensité dans ce fusible. Les caractéristiques détaillées des signaux à appliquer lors de la programmation sont consignées, pour chaque circuit et pour chaque fabricant, dans une base de données d'algorithmes du programmeur.

Pour chacun des 5828 fusibles de ce circuit, un '0' indique un fusible intact, un '1' un fusible programmé.

L'examen du début de la table précédente met en évidence un défaut majeur de cette technologie : la programmation détruit plus de fusibles qu'elle n'en conserve, et de loin. Cela se traduit par une mauvaise utilisation du silicium, un temps de programmation important (quelques secondes) et des contraintes thermiques sévères imposées au circuit lors de l'opération. Cette technologie n'est donc pas généralisable à des circuits dépassant quelques centaines de portes équivalentes.

Le lecteur averti aura peut-être remarqué, à la lecture de l'en-tête du fichier JEDEC, qu'en réalité le circuit précédent ne contient aucun fusible. Il s'agit en vérité d'un circuit CMOS à grille flottante, mais l'ancienne terminologie est restée.

**b) MOS à grille flottante**

Les transistors MOS sont des interrupteurs<sup>1</sup>, commandés par une charge électrique stockée sur leur électrode de grille. Si, en fonctionnement normal, cette grille est isolée, elle conserve sa charge éventuelle éternellement<sup>2</sup>. Il reste au fondeur à trouver un moyen de modifier cette charge, pour programmer l'état du transistor. Le dépôt d'une charge électrique sur la grille isolée d'un transistor fait appel à un phénomène connu sous le nom d'effet tunnel : un isolant très mince (une cinquantaine d'angströms,  $1 \text{ \AA} = 10^{-10} \text{ m}$ ) soumis à une différence de potentiel suffisamment grande (une dizaine de volts, supérieure aux 3,3 ou 5 volts des alimentations classiques) est parcouru par un courant de faible valeur, qui permet de déposer une

1. Quand on les utilise en tout ou rien, le régime source de courant contrôlée relève du monde des fonctions analogiques.

2. L'éternité en question est garantie durer plus de 20 ans.

charge électrique sur une électrode normalement isolée. Ce phénomène, réversible, permet de programmer et d'effacer une mémoire.

La figure 3-10 montre la structure du PLD élémentaire précédent, dans lequel les fusibles sont remplacés par des transistors à grille isolée (technologie FLASH).

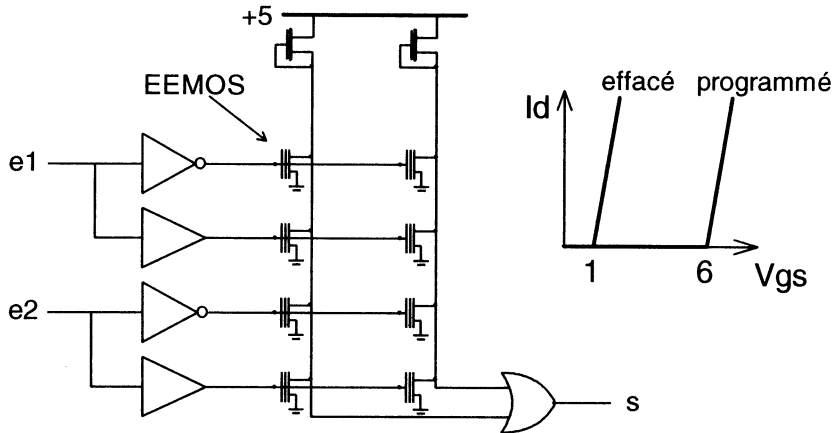


Figure 3-10 Pld simple à MOS.

Les transistors disposent de deux grilles, dont l'une est isolée. Une charge négative (des électrons) déposée sur cette dernière, modifie la tension de seuil du transistor commandé par la grille non isolée. Quand cette tension de seuil dépasse la tension d'alimentation, le transistor est toujours bloqué (interrupteur ouvert). Une variante (plus ancienne) de cette structure consiste à mettre deux transistors en série, l'un à grille isolée, l'autre normal. Le transistor à grille isolée est programmé pour être toujours conducteur ou toujours bloqué ; on retrouve exactement la fonction du fusible, la réversibilité en plus.

Le contrôle des dimensions géométriques des transistors permet actuellement d'obtenir des circuits fiables, programmables sous une dizaine de volts, reprogrammables à volonté (plusieurs centaines de fois), le tout électriquement.

Les puissances mises en jeu lors de la programmation sont suffisamment faibles pour que les surtensions nécessaires puissent être générées par les circuits eux-mêmes. Vu par l'utilisateur, le circuit devient alors programmable *in situ*, c'est-à-dire sans appareillage accessoire. Dans ces circuits, un automate auxiliaire gère les algorithmes de programmation et le dialogue avec le système de développement, *via* une liaison série.

### c) Mémoires statiques

Dans les circuits précédents, la programmation de l'état des interrupteurs, conservée en l'absence de tension d'alimentation, fait appel à un mode de fonctionnement électrique particulier. Dans les technologies à mémoire statique, l'état de chaque interrupteur est commandé par une cellule mémoire classique à quatre transistors (plus un transistor de programmation), dont le schéma de principe est celui de la figure 3-11.

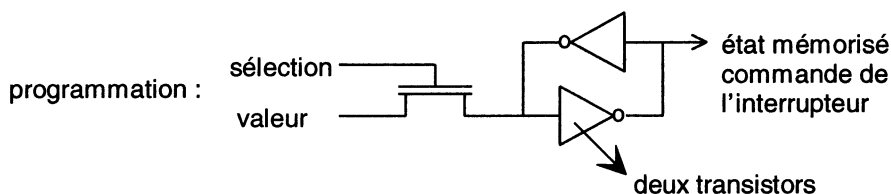


Figure 3-11 Cellule SRAM.

La modification de la configuration d'un circuit devient alors une opération logique quasi ordinaire, qui ne nécessite pas d'opération électrique spéciale. Ces circuits permettent des reconfigurations, partielles ou totales, en nombre illimité. Il est même envisageable de créer des fonctions dont certains paramètres sont modifiables en cours de fonctionnement, des filtres adaptatifs par exemple.

Le prix à payer pour cette souplesse est que les cellules SRAM doivent être rechargées à chaque mise sous tension et que chaque interrupteur occupe plusieurs transistors : l'interrupteur lui-même et les transistors de la cellule mémoire.

#### d) Antifusibles

L'inverse d'un fusible est un antifusible. Le principe est, à l'échelle microscopique, celui de la soudure électrique par points. Un point d'interconnexion est réalisé au croisement de deux pistes conductrices (métal ou semi-conducteur selon les procédés de fabrication), séparées par un isolant de faible épaisseur. Une surtension appliquée entre les deux pistes provoque un perçage définitif du diélectrique, ce qui établit la connexion.

Les points d'interconnexions ont un diamètre de l'ordre de la largeur d'une piste, c'est-à-dire de l'ordre du micron ; il est donc possible de prévoir un très grand nombre d'interconnexions programmables. La résistance du contact créé est très faible, de l'ordre d'une cinquantaine d'ohms (dix fois moins que celle d'un transistor MOS), d'où des retards liés aux interconnexions très faibles également.

Les circuits à antifusibles partagent, avec ceux à SRAM, le sommet de la gamme des circuits programmables en vitesse et en densité d'intégration.

Il est clair que ces circuits ne sont programmables qu'une fois.

### 3.2.2 Des architectures

Les différences de technologies se doublent de différences d'architectures. Nous tenterons ici de mettre en lumière des grands points de repère, sachant que toute classification a un côté un peu réducteur. La plupart des circuits complexes panachent les architectures.

#### a) Somme de produits

Toute fonction logique combinatoire peut être écrite comme somme de produits, nous avons évoqué ce point à propos des PLAs. La partie combinatoire d'un circuit programmable peut donc être construite en suivant cette démarche : chaque sortie est une fonction de toutes les entrées. Si la sortie se rapporte à un opérateur séquentiel,



les équations programmables calculent la valeur de la commande d'une bascule en fonction des entrées et des états de toutes les bascules du circuit : nous retrouvons l'architecture matérielle d'une machine d'états générique. Les PLDs de première génération suivaient ce principe.

La capacité de calcul de cette architecture est limitée par le nombre maximum de produits réunis dans la somme logique, et, dans une moindre mesure, par le nombre de facteurs de chaque produit. Les valeurs typiques sont respectivement de 16 et 44 pour un 22V10.

Très efficace pour la réalisation d'opérateurs relativement simples, cette architecture n'est pas directement généralisable à des circuits complexes : pour augmenter la capacité potentielle de calcul du circuit, il faut augmenter les dimensions des produits et des sommes logiques. Or même dans une fonction complexe, de nombreux sous-ensembles sont simples ; ces sous-ensembles monopoliseront inutilement une grande partie des potentialités du circuit<sup>1</sup>.

#### b) Cellules universelles interconnectées

L'autre approche, radicalement opposée, est de renoncer à la réduction en première forme normale des équations logiques. On divise le circuit en blocs logiques indépendants, interconnectés par des chemins de routage. Une fonction logique est récursivement décomposée en opérations plus simples, jusqu'à ce que les opérations élémentaires rentrent dans une cellule. La figure 3-12 en fournit un exemple.

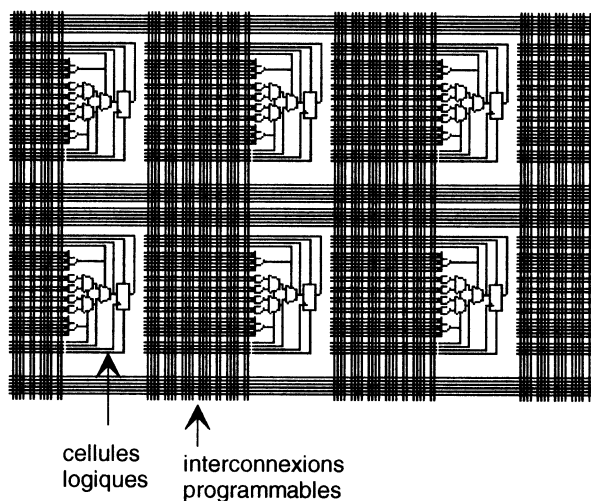


Figure 3-12 Cellules logiques interconnectées.

1. L'architecture du 22V10 contourne cette difficulté en utilisant des sommes de dimensions différentes (de 8 à 16). Mais cette solution impose des contraintes sur l'affectation des broches aux sorties d'une fonction : les broches centrales sont plus « puissantes » que les broches situées aux extrémités d'un boîtier DIL. Cela peut, par exemple, interdire la modification d'une fonction en conservant le câblage extérieur.

Les différences d'architectures entre les circuits concernent le compromis fait entre capacité de calcul de chaque cellule et possibilités d'interconnexions :

- Cellules de grande taille, à la limite l'équivalent d'un PLD classique, et interconnexions limitées. C'est schématiquement le choix fait pour les circuits CPLDs, en technologie FLASH.
- À l'autre extrême, cellules très petites (une bascule et un multiplexeur de commande), avec des ressources de routage importantes. C'est typiquement le choix fait dans les FPGAs à antifusibles, dont la figure 3-12 est un exemple.
- La solution intermédiaire est, sans doute, la plus répandue : les cellules comportent une ou deux bascules et des blocs de calcul combinatoires qui traitent de 6 à 10 entrées. Ces cellules sont optimisées pour accroître l'efficacité de traitement d'opérations courantes, comme le comptage ou l'arithmétique. Les circuits FPGAs à SRAM sont généralement associés à de telles cellules de taille moyenne.

### c) Cellules d'entrée-sortie

Dans les circuits programmables de première génération, les sorties étaient associées de façon rigide à des nœuds internes du circuit : résultat combinatoire, état d'une bascule.

Très vite est apparu l'intérêt de créer des macrocellules d'entrée-sortie pourvues d'une certaine capacité de reconfiguration. La figure 3-13 reprend le schéma de principe des cellules d'un PLD 22V10.

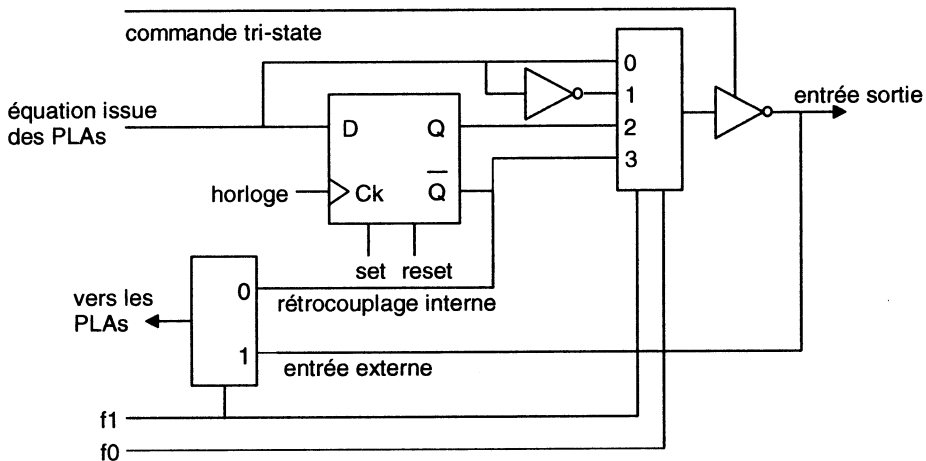


Figure 3-13 Macrocellule configurable.

Les deux fusibles f1 et f0 permettent de configurer la macrocellule en entrée-sortie combinatoire bidirectionnelle, complémentée ou non<sup>1</sup>, ou en sortie registre

1. Complémenter une sortie permet, dans certains cas, de simplifier les équations logiques.

trois états. Chaque sortie du circuit peut disposer de son propre mode, grâce aux vingt fusibles de configuration.

Les évolutions ultérieures rendent indépendantes les macrocellules et les broches du circuit, autorisant ainsi la création de bascules enterrées (*buried flip flops*) et d'entrées-sorties bidirectionnelles, quel que soit le mode, registre ou non, attaché à la sortie. Dans les architectures à cellules universelles interconnectées des FPGAs, les cellules d'entrée-sortie sont entièrement configurables et routables, au même titre que les cellules de calcul. Elles disposent de leurs propres bascules de synchronisation, en entrée et en sortie, indépendantes de celles des blocs logiques qui interviennent dans la fonction programmée.

#### e) Placement et routage

Le placement consiste à attacher des blocs de calcul aux opérateurs logiques d'une fonction et à choisir les broches d'entrée-sorties. Le routage consiste à créer les interconnexions nécessaires.

Pour les PLDs simples, le placement est relativement trivial et le routage inexistant. Les compilateurs génériques (i.e. indépendants du fondeur) effectuent très bien ces deux opérations.

Dès les CPLDs, et plus encore pour les FPGAs, ces deux opérations deviennent plus complexes et nécessitent un outil spécifique du fondeur, qui seul a les compétences nécessaires<sup>1</sup>. Le compilateur VHDL sert, dans ces cas, de frontal homogène qui traduit, après une première optimisation, la description VHDL dans un langage structurel adapté au logiciel spécifique<sup>2</sup>. Nous avons vu, à propos de la rétroannotation, que les outils des fondeurs fournissent en retour un modèle, VHDL ou VERILOG, du circuit généré qui prend en compte les temps de propagation internes.

### 3.2.4 Des techniques de programmation

Le placeur-routeur transforme la description structurelle du circuit en une table des fusibles consignée dans un fichier (JEDEC dans les cas simples, LOF, POF, etc., autrement). Pour la petite histoire, signalons que cette table peut contenir plusieurs centaines de milliers de bits, un par « fusible ».

Traditionnellement, la programmation du circuit, opération qui consiste à traduire la table des fusibles en une configuration matérielle, se faisait au moyen d'un programmeur, appareil capable de générer les séquences et les surtensions nécessaires. La tendance actuelle est de supprimer cette étape de manipulation intermédiaire, manipulation d'autant plus malaisée que l'augmentation de la complexité des boîtiers va de pair avec celle des circuits. Autant il était simple de concevoir des

1. Pour la simple raison qu'il est seul à connaître ses circuits dans leurs moindres détails.

2. Une certaine portabilité demeure, même à ce niveau. Il existe des formats de fichiers communs à plusieurs fondeurs : les fichiers PLA ou, plus souvent, des fichiers dans un langage symbolique, EDIF pour *electronic data interchange format for net-lists*. Il s'agit d'un langage de description structurelle, qui ressemble un peu à LISP, compris par la majorité des systèmes de CAO.

supports à force d'insertion nulle pour des boîtiers DIL (*dual in line*) de 20 à 40 broches espacées de 2,54 mm, autant il est difficile et coûteux de réaliser l'équivalent pour des PGA (*pin grid array*) et autres BGA (*ball grid array*), de 200 à plus de 300 broches réparties sur toute la surface du boîtier, sans parler des boîtiers miniaturisés, au pas de 0,65 mm, destinés au montage en surface.

Une difficulté du même ordre se rencontre pour le test : il est devenu quasi impossible d'accéder, par des moyens traditionnels tels que les pointes de contact d'une « planche à clous », aux équipotentielles d'une carte. De toute façon, les équipotentielles du circuit imprimé ne représentent plus qu'une faible proportion des nœuds du schéma global : un circuit de 250 broches peut contenir 2 500 bascules.

#### a) Trois modes : fonctionnement normal, programmation et test

Fonctionnement normal, programmation et test : l'idée s'est imposée d'incorporer ces trois modes de fonctionnement dans les circuits eux-mêmes, comme partie intégrante de leur architecture. Pour le test de cartes, une norme existe : le standard IEEE 1149.1, plus connu sous le nom de *boundary scan* du consortium JTAG (*join test action group*). Face à la quasi-impossibilité de tester de l'extérieur les cartes multicouches avec des composants montés en surface, un mode de test a été défini, pour les VLSI numériques. Ce mode de test fait appel à une machine d'états, intégrée dans tous les circuits compatibles JTAG, qui utilise cinq broches dédiées :

- Tck, une entrée d'horloge dédiée au test, différente de l'horloge du reste du circuit.
- Tms, une entrée de mode qui pilote l'automate de test.
- Tdi, une entrée série.
- Tdo, une sortie série.
- Trst (optionnelle), une entrée de réinitialisation asynchrone de l'automate.

L'utilisation première de ce sous-ensemble de test est la vérification des connexions d'une carte. Quand le mode de test est activé, *via* des commandes *ad hoc* sur les entrées Tms et Trst, le fonctionnement normal du circuit est inhibé. Les broches du circuit sont connectées à des cellules d'entrée-sortie dédiées au test<sup>1</sup>, chaque cellule est capable de piloter une broche en sortie et de capturer les données d'entrée, conformément au schéma de principe de la figure 3-14.

Toutes les cellules de test sont connectées en un registre à décalage, tant à l'intérieur d'un circuit qu'entre les circuits, constituant ainsi une chaîne de données, accessible en série, qui parcourt l'ensemble des broches de tous les circuits compatibles JTAG d'une carte. Les opérations de test sont programmées *via* des commandes passées aux automates et des données entrées en série. Les résultats des tests sont récupérables par la dernière sortie série.

1. Typiquement, une broche d'entrée-sortie bidirectionnelle est pilotée par 6 bascules : un couple en entrée, un couple en sortie et un couple en commande de trois états. Les bascules par paires permettent de décaler les données tout en mémorisant la configuration précédente de chaque broche.

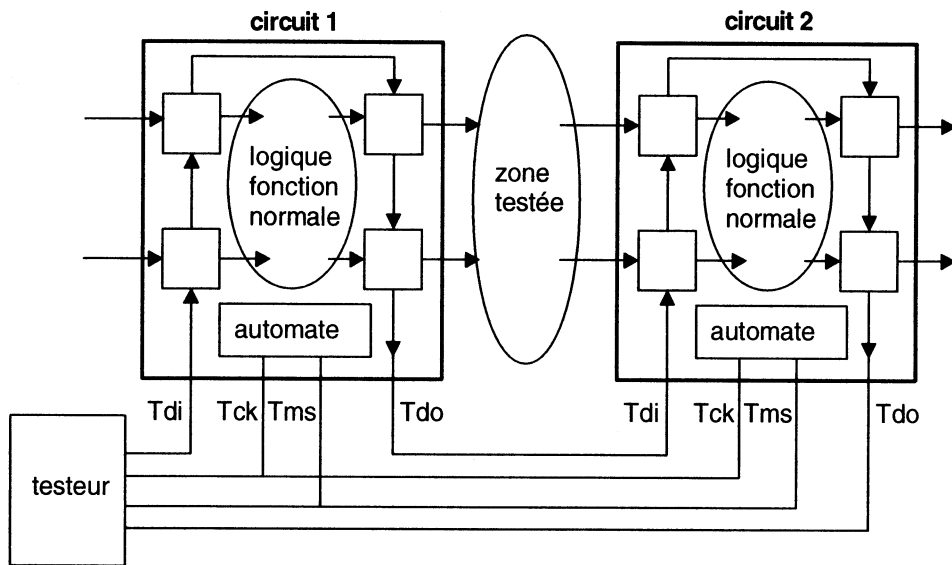


Figure 3-14 Boundary scan.

Les automates de test permettent d'autres vérifications que celles des connexions : il est possible de les utiliser pour appliquer des vecteurs de test internes aux circuits, par exemple. C'est souvent de cette façon que sont effectués certains des tests à la fabrication. L'idée était séduisante d'utiliser la même structure pour configurer les circuits programmables. C'est ce qui est en train de se faire : la plupart des fabricants proposent des solutions plus ou moins dérivées de JTAG pour éviter à l'utilisateur d'avoir recours à un appareillage extérieur<sup>1</sup>.

#### b) Programmables in situ

Les circuits programmables *in situ* se développent dans le monde des PLDs et CPLDs en technologie FLASH. Du simple 22V10, à des composants de plus de 10000 portes équivalentes et 400 bascules (LATTICE, par exemple), il est possible de programmer (et de modifier) l'ensemble d'une carte, sans démontage, à partir d'un port parallèle de PC.

Les technologies FLASH conservent leur configuration en l'absence d'alimentation.

#### » Reconfigurables dynamiquement

Les FPGAs à cellules SRAM offrent des possibilités multiples de chargement de la mémoire de configuration :

- Chargement automatique, à chaque mise sous tension, des données stockées dans une mémoire PROM. Les données peuvent être transmises en série, en utilisant

1. Les pionniers en la matière furent les sociétés XILINX pour les technologies SRAM et LATTICE pour les technologies FLASH.

peu de broches du circuit, ou en parallèle octet par octet, ce qui accélère la phase de configuration mais utilise, temporairement du moins, plus de broches du circuit. Plusieurs circuits d'une même carte peuvent être configurés en coopération, leurs automates de chargement assurent un passage en mode normal coordonné, ce qui est évidemment souhaitable.

- Chargement, en série ou en parallèle, à partir d'un processeur maître. Ce type de structure autorise la modification rapide des configurations en cours de fonctionnement. Cette possibilité est intéressante, par exemple, en traitement de signal.

### 3.3 PLDS, CPLDS, FPGAS : QUEL CIRCUIT CHOISIR ?

Dans le monde des circuits numériques les chiffres évoluent très vite, beaucoup plus vite que les concepts. Cette impression de mouvement permanent est accentuée par les effets d'annonce des fabricants et par l'usage systématique de la publicité comparative, très en vogue dans ce domaine.

Il semble que doivent se maintenir trois grandes familles :

- Les PLDs et CPLDs en technologie FLASH, utilisant une architecture somme de produits. La tendance est à la généralisation de la programmation *in situ*, rendant inutiles les programmeurs sophistiqués. Réservés à des fonctions simples ou moyennement complexes, ces circuits sont rapides (jusqu'à environ 200 MHz) et leurs caractéristiques temporelles sont pratiquement indépendantes de la fonction réalisée. Les valeurs de fréquence maximum de fonctionnement de la notice sont directement applicables.
- Les FPGAs à SRAM, utilisant une architecture cellulaire. Proposés pratiquement par tous les fabricants, ils couvrent une gamme extrêmement large de produits, tant en densités qu'en vitesses. Reprogrammables indéfiniment, ils sont devenus reconfigurables rapidement (200 ns par cellule), en totalité ou partiellement.
- Les FPGAs à antifusibles, utilisant une architecture cellulaire à granularité fine. Ces circuits tendent à remplacer une bonne partie des ASICs prédiffusés. Programmables une fois, ils présentent l'avantage d'une très grande routabilité, d'où une bonne occupation de la surface du circuit. Leur configuration est absolument immuable et disponible sans aucun délai après la mise sous tension ; c'est un avantage parfois incontournable.

#### 3.3.1 Critères de performances

Outre la technologie de programmation, capacité et vitesse sont les maîtres mots pour comparer deux circuits. Mais quelle capacité, et quelle vitesse ?

##### a) Puissance de calcul

Les premiers chiffres accessibles concernent les nombres d'opérateurs utilisables.

##### ➤ Nombre de portes équivalentes

Le nombre de portes est sans doute l'argument le plus utilisé dans les effets d'annonce. En 2000 la barrière des 250 000 portes est largement franchie. Plus délicate

est l'estimation du nombre de portes qui seront inutilisées dans une application, donc le nombre réellement utile de portes.

#### ➤ Nombre de cellules

Le nombre de cellules est un chiffre plus facilement interprétable : le constructeur du circuit a optimisé son architecture, pour rendre chaque cellule capable de traiter à peu près tout calcul dont la complexité est en relation avec le nombre de bascules qu'elle contient (une ou deux suivant les architectures). Trois repères chiffrés : un 22V10 contient 10 bascules, la famille des CPLDs va de 32 bascules à quelques centaines et celle des FPGAs s'étend d'une centaine à quelques milliers.

Dans les circuits à architectures cellulaires, il est souvent très rentable d'augmenter le nombre de bascules si cela permet d'alléger les blocs combinatoires (*pipe line*, codages *one hot*, etc.).

#### ➤ Nombre d'entrée-sorties

Le nombre de ports de communication entre l'intérieur et l'extérieur d'un circuit peut varier dans un rapport deux, pour la même architecture interne, en fonction du boîtier choisi. Les chiffres vont de quelques dizaines à quelques centaines de broches d'entrée-sorties..

#### ➤ Capacité mémoire

Les FPGAs à SRAM contiennent des mémoires pour stocker leur configuration. La plupart des familles récentes offrent à l'utilisateur la possibilité d'utiliser certaines de ces mémoires en tant que telles. Par exemple, la famille 4000 de XILINX permet d'utiliser les mémoires de configuration d'une cellule pour stocker 32 bits de données ; la cellule correspondante n'est évidemment plus disponible comme opérateur logique. Les capacités de mémorisation atteignent quelques dizaines de kilobits.

#### ➤ Routabilité

Placement et routage sont intimement liés, et le souhait évident de l'utilisateur est que ces opérations soient aussi automatiques que possible. Le critère premier de routabilité est l'indépendance entre la fonction et le brochage. Certains circuits (mais pas tous) garantissent une routabilité complète : toute fonction intégrable dans le circuit pourra être modifiée sans modification du câblage externe.

Le routage influe sur les performances dynamiques de la fonction finale. La politique généralement adoptée est de prévoir des interconnexions hiérarchisées : les cellules sont regroupées en grappes (d'une ou quelques dizaines) fortement interconnectées, des pistes de communication reliant les grappes entre elles. Les interconnexions locales n'ont que peu d'influence sur les temps de calcul, contrairement aux interconnexions distantes dont l'effet est notable.

*A priori* c'est au placeur-routeur que revient la gestion de ces interconnexions ; à condition que le programmeur ne lui complique pas inutilement la tâche. Un optimiseur a toujours du mal à découper des blocs de grandes tailles, il lui est beaucoup plus simple de placer des objets de petites dimensions. En VHDL cela s'appelle

construction hiérarchique ; un ensemble complexe doit être conçu comme l'assemblage d'unités de conceptions aussi simples que possibles.

### b) Vitesse de fonctionnement

Nous avons vu, à propos de la rétroannotation, que les comportements dynamiques des FPGAs et des PLDs simples présentent des différences marquantes. Les premiers ont un comportement prévisible, indépendamment de la fonction programmée ; les limites des seconds dépendent de la fonction, du placement et du routage. Une difficulté de jeunesse des FPGAs a été la non-reproductibilité des performances dynamiques en cas de modification, même mineure, du contenu d'un circuit. Les logiciels d'optimisation et les progrès des architectures internes ont pratiquement supprimé ce défaut ; mais il reste que seule une analyse et une simulation postsynthèse, qui prend en compte les paramètres dynamiques des cellules, permet réellement de prévoir les limites de fonctionnement d'un circuit.

#### ► Modèle général de détermination de fmax

Le modèle général de détermination de la fréquence maximum d'un opérateur séquentiel prend en compte les retards dans les circuits et les règles concernant les instants de changement des entrées vis-à-vis des fronts actifs de l'horloge. La figure 3-15 définit les temps les plus importants :  $t_{p1}$  pour des temps de propagation et  $t_{su}$  pour le temps de prépositionnement d'une bascule.

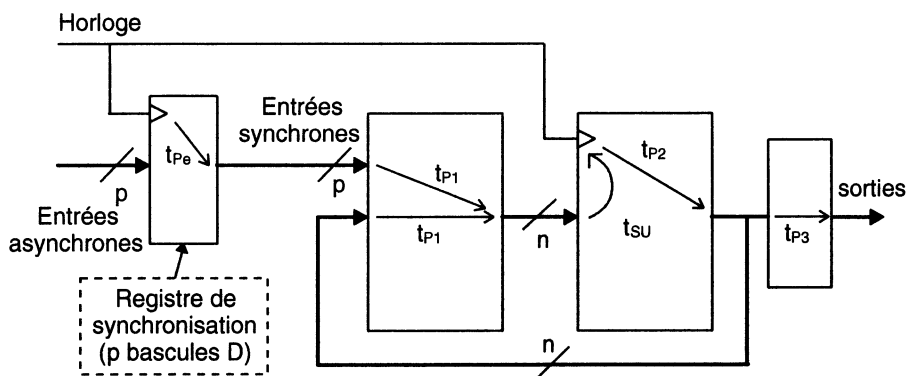


Figure 3-15 Modèle de calcul de la fréquence maximum.

La fréquence maximum de fonctionnement interne est donnée par :

$$F_{int} = 1/(t_{p2} + t_{p1} + t_{su})$$

Pour le calcul de la fréquence maximum externe, il convient de rajouter, dans la formule précédente, le temps de propagation des cellules de sortie :

$$F_{ext} = 1/(t_{p3} + t_{p2} + t_{p1} + t_{su})$$



Dans un FPGA le routeur analyse le schéma généré et en déduit les différents temps de propagation, à partir d'un modèle des cellules élémentaires du circuit.

Dans le cas des PLDs simples et de beaucoup (pas tous) des CPLDs, les notices fournissent directement les valeurs des fréquences maximum et/ou des temps de retard et de prépositionnement entre les signaux appliqués aux broches du circuit et les fronts de l'horloge.

#### » Style de programmation et performances

Pour l'auteur d'un programme VHDL, quelques guides de programmation sont utiles :

- Réfléchir au codage des états, dans la conception des machines d'états. Les sorties directes du registre d'état sont préférables ; les codes *one hot* sont très efficaces dans les FPGAs. Nous avons évoqué ces points précédemment.
- Subdiviser les blocs de calcul combinatoires en tranches séparées par des registres ; autrement dit, penser aux architectures *pipe line*. Ces architectures génèrent un retard global de plusieurs périodes d'horloge, mais permettent d'obtenir des flots de données rapides<sup>1</sup>.
- Savoir que les bibliothèques des fondeurs sont riches en modules structurels optimisés en fonction du circuit cible : les modules LPM (*Library of Parameterized Modules*).
- Les synthétiseurs infèrent automatiquement des modules LPM, à partir de descriptions comportementales de haut niveau, sous réserve que le programmeur respecte certaines règles d'écriture ou indique explicitement qu'il souhaite utiliser les librairies correspondantes. La notice de tous les compilateurs explique la démarche à suivre.

#### c) Consommation

Les premiers circuits programmables avaient plutôt mauvaise réputation sur ce point. Tous les circuits actuels ont fait d'importants progrès en direction de consommations plus faibles.

#### » Le compromis vitesse consommation

Règle générale des circuits numériques, encore plus vraie dans le monde des technologies MOS que dans celui des technologies bipolaires : pour aller vite il faut de la puissance. Les notices fournissent communément des courbes de consommation pour des éléments classiques, comme un compteur synchrone 16 bits, en fonction de la fréquence d'horloge. Le passage à 3,3 V des tensions d'alimentation permet une économie non négligeable de puissance, pour les mêmes valeurs de courant.

Les cellules de certains circuits sont programmables en deux modes : faible consommation ou vitesse maximum (bit *turbo*). Le gain de vitesse se paye par une consommation nettement plus élevée (pratiquement un facteur 2 pour un EPM7032 cadencé à 60 Mhz, par exemple).

---

1. L'image classique est le principe de la fabrication des voitures à la chaîne : même si une voiture sort toutes les dix minutes, il faut plus de dix minutes pour fabriquer une voiture prise isolément. Le débit est très supérieur à l'inverse du temps de fabrication d'une seule voiture.

De façon générale, le courant moyen consommé par un circuit est de la forme :

$$I_{CC} = I_{CC0} + k \times n_L \times F + n_S \times C_S \times \Delta V \times F/2$$

Où  $I_{CC0}$  représente le courant statique consommé au repos,  $n_L$  le nombre moyen de cellules logiques qui commutent simultanément à chaque front d'horloge,  $n_S$  le nombre moyen de sorties qui commutent à chaque front d'horloge,  $C_S$  la capacité de charge moyenne des sorties,  $\Delta V$  l'excursion de la tension de sortie,  $F$  la fréquence d'horloge et  $k$  un coefficient de consommation par cellule par hertz.

#### ➤ Quelques chiffres

Un ordre de grandeur du paramètre  $k$  précédent est, pour un FPGA de la famille FLEX 8000 d'ALTERA, de 150  $\mu A$  par MHz et par cellule.

Un circuit cadencé à 50 MHz, dans lequel 100 cellules commutent, en moyenne, à chaque front d'horloge, consomme, sans charge extérieure, un courant moyen de l'ordre de 750 mA. Ce qui est loin d'être négligeable.

À titre de confrontation, la valeur précédente doit être comparée à la consommation de 50 compteurs binaires<sup>1</sup>. Un compteur binaire de la famille TTL-AS (il faut prendre des circuits de vitesses comparables) consomme 35 mA. Les chiffres parlent d'eux-mêmes.

Toujours dans le même ordre, un PLD 22V10 rapide, 10 cellules, consomme un courant de l'ordre de 100 mA. Dans ce dernier cas, la fonction programmée et la fréquence d'horloge n'ont qu'une incidence faible sur le courant consommé.

#### d) L'organisation PREP

Nombre de portes, de cellules, de bascules, fréquence maximum, dans quelle condition ? Avec quel logiciel ? Les comparaisons ne sont pas simples.

Le consortium PREP (*programmable electronics performance corporation*) regroupait jusqu'en 1996 la plupart des fabricants de circuits programmables. Cet organisme a défini un ensemble de neuf applications, typiques de l'usage courant des circuits programmables, qui servent de test à la fois pour les circuits et le système de développement associé. Les fruits de la confrontation à ce *benchmark* sont fournis pour la plupart des CPLDs et FPGAs. Ces résultats contiennent des informations de vitesse, fréquences maximums interne et externe pour chaque test, et de capacité, nombre moyen d'exemplaire de chaque test que l'on peut instancier dans un circuit.

#### ➤ Des applications « types »

Les 9 épreuves de test sont :

- *Datapath* : un chemin de données, sur un octet, franchit dans l'ordre un multiplexeur 4 vers 1, un registre tampon et un registre à décalage arithmétique (avec

1. Dans un compteur binaire deux cellules commutent, en moyenne, à chaque période d'horloge. Nous laissons au lecteur le soin de le démontrer.

extension de signe). Le schéma ne comporte pratiquement pas de calcul entre les bascules des registres ; les fréquences maximum obtenues sont à peu de choses près celles des circuits en boucle ouverte. Le nombre de vecteurs d'entrée sollicite beaucoup les ressources de routage.

- *Counter timer* : Un compteur 8 bits à chargement parallèle parcourt un cycle défini par une valeur de chargement et une valeur finale. Un comparateur provoque le rechargement du compteur quand il a atteint la valeur finale. Le schéma comporte, outre le compteur, deux registres, un comparateur et un multiplexeur, le tout sur un octet. Le fonctionnement place le comparateur dans la boucle de commande du compteur, limitant par là sa fréquence maximum de fonctionnement.
- *Small state machine* : petite machine d'états, 8 états, 8 entrées, 8 sorties. Beaucoup de CPLDs arrivent à la faire fonctionner à leur fréquence maximum, les résultats sont plus variables pour les FPGAs.
- *Large state machine* : machine à 16 états, 8 entrées et 8 sorties. La plupart des CPLDs doivent abandonner leur fréquence de fonctionnement maximum : le nombre de variables est trop grand pour autoriser les calculs en une seule passe dans la logique combinatoire.
- *Arithmetic* : un multiplieur de deux nombres de 4 bits, résultat sur 8, suivi par un additionneur accumulateur sur 8 bits. La structure en « somme de produits » des CPLDs les rend très inefficaces dans les problèmes d'arithmétique. Les FPGAs montrent une supériorité architecturale nette face à ces problèmes.
- *Accumulateur* : accumulateur-additionneur sur 16 bits. Un additionneur de deux nombres de 16 bits est suivi par un registre dont le contenu est pris comme l'un des opérandes de l'addition. Ce test génère un schéma moins complexe que le précédent.
- *16-bit counter* : un classique compteur 16 bits, à chargement parallèle synchrone et remise à zéro asynchrone. C'est un peu un test de vitesse pure dans une application standard.
- *16-bit prescaled counter* : compteur 16 bits à prédiviseur synchrone. Pour accélérer le comptage, une technique consiste à traiter à part l'étage de poids faible d'un compteur, quitte à perdre la possibilité d'effectuer le chargement parallèle en un seul cycle. Pour les CPLDs il n'y a aucune différence avec l'épreuve précédente ; pour les FPGAs l'architecture en petites cellules conduit à une accélération nette du fonctionnement.
- *Décodeur d'adresses* : un décodeur d'adresse génère 8 signaux de décodage mémorisés dans un registre à partir d'un signal d'entrée sur 16 bits ; il découpe ainsi l'espace d'adresses en huit pages. Le traitement de ce schéma favorise les circuits qui disposent de portes ET à grand nombre d'entrées. L'architecture CPLD se prête mieux à cette épreuve que celle des FPGAs.

#### ➤ À chacun son interprétation

Les résultats à ces épreuves sont généralement présentés sous forme de tableaux comparatifs. Chaque constructeur veillant, évidemment, à citer les résultats de la concurrence qui illustrent sa propre supériorité.

Une analyse générale, proposée par beaucoup de fabricants, consiste à calculer pour chaque circuit les moyennes des fréquences maximums de fonctionnement, et des nombres d'instances de chaque épreuve implantable dans le circuit. Ces deux chiffres fournissent une information globale de vitesse et une information globale de capacité. Sans entrer dans des comparaisons chiffrées qui ne valent qu'à un instant donné, il est intéressant de délimiter dans le plan fréquence/capacité les zones de prédilection des différentes catégories de circuits programmables. C'est le sens de la figure 3-16.

Les petits circuits sont incapables de contenir les épreuves PREP, ils se concentrent à une capacité moyenne nulle. Nous les avons malgré tout placés dans ce plan, bien que la comparaison entre des moyennes d'un côté et une information ponctuelle de l'autre soit un peu trompeuse<sup>1</sup>.

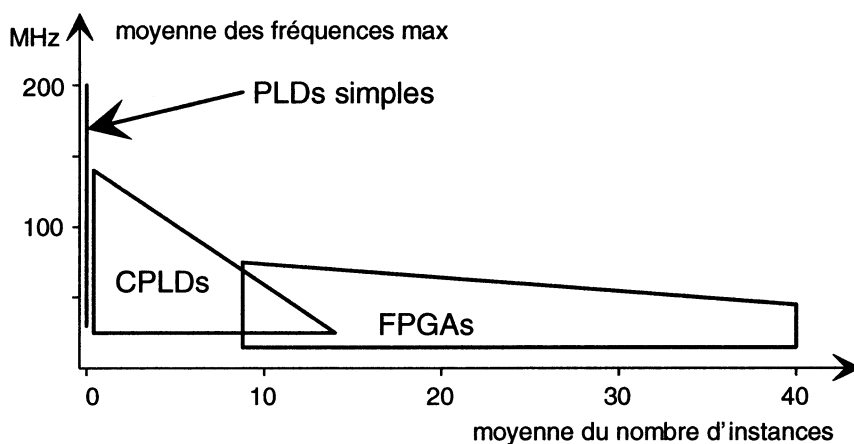


Figure 3-16 Moyennes des tests PREP.

### 3.3.2 Le rôle du « fitter »

Nous avons un peu exploré les aspects matériels des circuits programmables. Terminons en retrouvant, à travers quelques remarques, l'aspect logiciel des choses. Un circuit n'est rien sans son logiciel de développement ; un résultat surprenant aux épreuves PREP nous en fournit un exemple.

#### a) « Mise à plat » ou logique multicouches ?

L'une de ces épreuves concerne l'arithmétique : pour cette épreuve un CPLD qui n'est ni moins performant que son concurrent direct, ni plus petit, s'est vu attribuer, en 1995, la note zéro en fréquence maximum de fonctionnement. Un zéro dans une moyenne, cela pèse lourd. La règle est la synthèse automatique, optimisation comprise.

1. Il n'est pas difficile de trouver dans un *data book* un exemple particulier de montage avec lequel un FPGA dépasse très largement les 200 MHz.

Le *fitter* du fondeur concerné s'est vraisemblablement fourvoyé dans la mise à plat, sous forme somme de produits logiques, des opérateurs arithmétiques. Nous avons eu l'occasion d'évoquer le caractère explosif de cette démarche.

La bonne approche était, pour cet exemple, de conserver plus de couches logiques, au détriment de la vitesse.

Les performances des FPGAs ne se dégradent que très progressivement quand la complexité d'un schéma augmente ; cette faculté est liée à leur architecture à granularité fine, qui impose de toute façon de passer à des structures multicouches, même pour des fonctions combinatoires de complexité moyenne. Les constructeurs de circuits ont optimisé les passages de retenues d'une cellule à l'autre, qui autorisent des structures de propagation des retenues sans trop ralentir le système.

Certains logiciels donnent à l'utilisateur le loisir de régler manuellement le seuil de dédoublement d'équations logiques trop larges. Il est possible de spécifier le nombre maximum de facteurs dans un produit et le nombre maximum de termes dans une somme, par exemple. Ce genre de réglages manuels peut, bien qu'un peu délicat à manipuler, donner de bons résultats quand les choix automatiques ne conviennent plus.

#### *b) Surface ou vitesse*

Les options de réglage standard d'un *fitter* permettent de privilégier la surface (de silicium) ou la vitesse. Les deux choix sont, en effet, souvent contradictoires : pour diminuer la surface il faut augmenter le nombre de couches, ce qui pénalise la vitesse<sup>1</sup>.

#### *c) Librairies de macrofonctions*

Rappelons l'importance des librairies de modules optimisés en fonction du circuit cible. L'idéal est qu'elles soient explorées automatiquement par l'analyseur de code VHDL, mais cette recherche automatique suppose que le code source ne brouille pas les cartes.

#### *d) Le meilleur des compilateurs ne peut donner que ce que le circuit possède*

Ultime remarque : le meilleur des compilateurs ne peut qu'organiser ce qui préexiste dans un circuit, il ne crée rien. Trivialement, tenter d'implanter un compteur 16 bits dans un 22V10 est un objectif inaccessible. Cet exemple en fera sourire plus d'un ; Transposé à un circuit de compression de la parole, à implanter dans un circuit plus conséquent, le problème reste le même ; même si l'analyse de faisabilité est plus ardue, elle doit pourtant être poursuivie.

---

1. La situation réelle est un peu plus complexe : quand on diminue le nombre de couches logiques, la sortance imposée aux opérateurs augmente. Dans les technologies MOS les temps de propagation de ces opérateurs dépendent beaucoup de leurs capacités de charge, donc du nombre d'entrées qu'ils doivent commander. Les *fitter* contrôlent également ce type de contraintes.

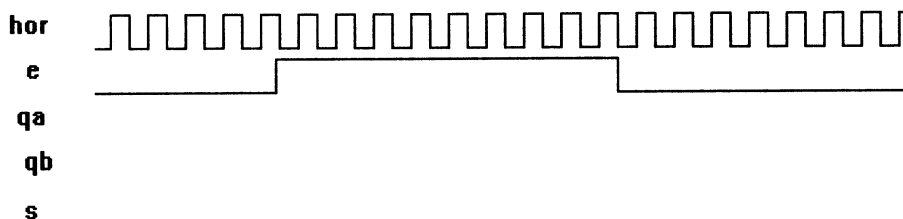
# Exercices

## 1. Signaux et processus

On considère le programme ci-dessous :

```
entity transitm is
  port (  hor, e : in bit ;
         s : out bit );
end transitm ;
architecture quasi_struct of transitm is
  signal qa, qb : bit ;
begin
  s <= qa xor qb ;
  schem : process
  begin
    wait until hor = '1' ;
    qa <= e ;
    qb <= qa ;
  end process schem ;
end quasi_struct ;
```

- Déduire de ce programme, par une construction méthodique, un schéma (bascules et portes logiques).
- Compléter le chronogramme ci-dessous.



## 2. Du code VHDL au schéma

On considère le programme VHDL suivant qui décrit le fonctionnement d'une bascule :

```
entity basc is
  port ( T,hor,init : in bit;
         s : out bit);
end basc;

architecture primitive of basc is
  signal etat : bit;
begin
  s <= etat ;
  process
  begin
    wait until (hor = '1') ;
    if(init = '0') then
      etat <= '1';
    elsif(T = '0') then
      etat <= not etat;
    end if;
  end process;
end primitive;
```

- À quoi reconnaît-on qu'il s'agit d'un circuit séquentiel synchrone ?
- La commande « init » est-elle synchrone ou asynchrone ? (justifier)
- Établir le diagramme de transitions de cette bascule.
- Dédurre du diagramme précédent les équations logiques et le schéma d'une réalisation avec une bascule D.
- Modifier le programme précédent pour qu'il rajoute à la bascule une commande raz, de remise à zéro, asynchrone.
- Transformer le programme pour qu'il utilise le type std\_logic. Rajouter une commande oe de contrôle qui fasse passer la sortie en haute impédance si oe = '0'.

## 3. Synchronisation interprocessus

On considère le programme VHDL ci-dessous :

```
entity couple is
  port ( hor,raz : in bit ;
         cpt : out integer range 0 to 3 ) ;
end couple ;

architecture deux_proc of couple is
  signal compte : integer range 0 to 3 ;
  signal attente : bit ;
begin
  cpt <= compte ;

  compteur : process (hor, raz)
```

```

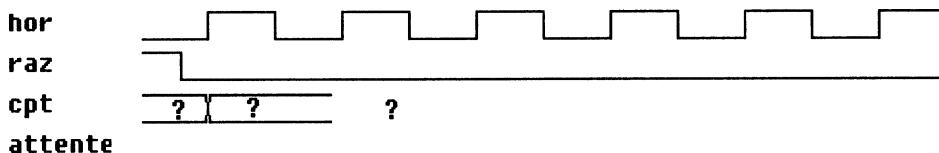
begin
  if raz = '1' then
    compte <= 0 ;
  elsif hor'event and hor = '1' then
    case compte is
      when 2 => if attente = '1' then
                  compte <= compte + 1 ;
                end if ;
      when others => compte <= (compte + 1) mod 4 ;
    end case ;
  end if ;
end process compteur ;

retard : process
begin
  wait until hor = '1' ;
  if compte = 2 then
    attente <= '1' ;
  else
    attente <= '0' ;
  end if ;
end process retard ;
end deux_proc ;

```

Questions : (toutes les réponses doivent être obtenues par une analyse du programme)

- Établir un schéma synoptique du circuit décrit. Dans cette question on ne demande pas les détails, seuls doivent figurer des blocs et les signaux nommés dans le programme.
- Combien de bascules ce programme génère-t-il ?
- Certaines de ces bascules ont un fonctionnement entièrement synchrone, d'autres possèdent une commande asynchrone de l'horloge. Préciser ce point.
- Compléter le chronogramme ci-dessous et commenter le résultat.



- Représenter le fonctionnement synchrone de chaque processus par un diagramme de transitions.
- Dédire des diagrammes précédents les équations de commandes des bascules avec des bascules D.

Représenter les logigrammes correspondants (portes et bascules pourvues des commandes asynchrones éventuellement nécessaires).



- Cette question concerne le schéma obtenu avec des bascules D.  
Les paramètres dynamiques des portes et bascules sont les suivants :
 

temps de propagation	$T_p \leq 10 \text{ ns}$
temps de maintien	$T_{hold} = 0$
temps de prépositionnement	$T_{setup} = 15 \text{ ns}$

 Représenter le fonctionnement du système pendant deux périodes caractéristiques de l'horloge, en faisant figurer ces temps.  
Quelle est la fréquence maximum de fonctionnement du montage ?
- Quels sont les points à regarder dans la notice d'un circuit programmable que l'on voudrait utiliser pour réaliser le système ?

#### 4. Une petite fonction

On considère la fonction suivante :

```
function incremente(e : in bit_vector) return bit_vector is
  variable s : bit_vector(e'left downto 0) ;
  variable c : bit ;
begin
  c := e(0) ;
  s(0) := not c ;
  for i in 1 to e'left loop
    s(i) := c xor e(i) ;
    c := c and e(i) ;
  end loop ;
  return s ;
end incremente ;
```

- Dans quelles parties d'un programme VHDL peut-on insérer un tel code ? (deux réponses au moins)
- Représenter le schéma généré par cette fonction si on l'applique à un vecteur de trois éléments binaires.
- Quelle est l'opération réalisée ?
- Utiliser cette fonction pour réaliser un compteur binaire synchrone 16 bits qui compte à chaque front montant d'horloge.
- Modifier le compteur précédent pour lui adjoindre une commande raz de remise à zéro synchrone et une commande d'autorisation de comptage en.
- Même question pour une commande raz asynchrone de l'horloge.
- Proposer une fonction qui réalise un décalage à droite d'un vecteur et insère à gauche un élément binaire passé en argument. L'utiliser pour modéliser un registre à décalage synchrone à entrée série et sortie parallèle.

#### 5. Procédures

La boîte à outils du banc de test étudié au paragraphe 2.7.7 utilise une description structurelle.

- Reprendre la même idée de modularité en remplaçant les composants par des procédures.
- Discuter les avantages et inconvénients des deux approches.

## 6. Modélisation

Certains simulateurs pré-VHDL utilisaient un type logique multivalué à 12 états pour modéliser les sorties non standard et les conflits. Un état est obtenu par l'association d'un niveau et d'une force.

Trois niveaux logiques sont possibles :

- '0', '1' et 'X' (inconnu).

Quatre valeurs de forces définissent les résolutions de conflits :

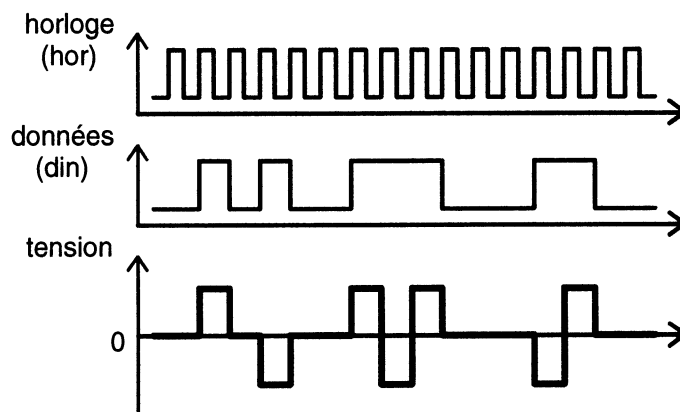
- 'S' pour forte (*strong*), 'W' pour faible (*weak*),
- 'Z' pour haute impédance et 'X' pour inconnue.

La valeur d'une équipotentielle est l'association d'un niveau et d'une force. On se propose de créer un type et les opérations de base associées qui permette d'utiliser cette modélisation (on utilisera un paquetage).

- Créer un type de base sous forme de record contenant des types énumérés.
- Créer, en suivant la démarche présentée au paragraphe 2.7.3, un type résolu dérivé, au moyen d'une table de résolution et d'une fonction associée.
- Créer des fonctions et des opérateurs logiques élémentaires qui permettent de modéliser des opérateurs logiques courants.
- Ces nouveaux types sont-ils synthétisables ?

## 7. Exercice de synthèse : codeur AMI

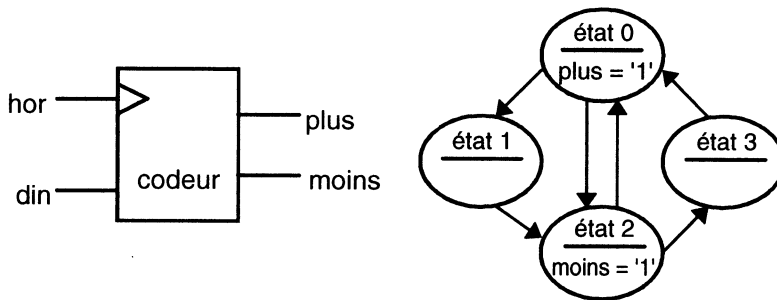
Dans les transmissions téléphoniques à grande distance, les informations transitent sous forme numérique, transmises en série (un bit à la fois), au rythme d'une horloge. Le code binaire utilisé est transformé en un code à 3 niveaux de tension sur la ligne (câble coaxial, par exemple) :



- un ZÉRO logique correspond toujours à une tension nulle,
- les niveaux logiques UN sont représentés par des impulsions, qui durent une période de l'horloge de transmission, alternativement positives et négatives, d'où le nom du code. On notera que le système doit se « souvenir » de la polarité de la dernière impulsion transmise pour fonctionner correctement.

La création des impulsions alternées passe par un changement de code : le codeur reçoit l'horloge d'émission, hor, et les données à transmettre, din. Il fournit en sortie deux signaux binaires que nous nommerons plus et moins, générés suivant l'algorithme ci-dessous :

- si  $din = '0'$  : plus = '0', moins = '0' ;
- si  $din = '1'$  : plus = '1', moins = '0' ou plus = '0', moins = '1', en alternance.



On se propose d'étudier plusieurs solutions pour réaliser ce codeur.

L'idée générale est de réaliser une machine synchrone à quatre états, conformément à la figure ci-dessus.

L'ébauche de diagramme de transitions proposé est évidemment incomplète.

- Compléter le diagramme de transitions proposé, en indiquant les conditions de transitions et de maintiens éventuels.
- Quel est le nombre minimum de bascules nécessaires à la réalisation du codeur ?
- Proposer un codage des états et une solution complète du problème qui utilise ce nombre minimum de bascules. Proposer un programme VHDL qui réponde au problème.
- On souhaite que les sorties du codeur soient issues directement de bascules du registre d'état. Proposer une solution en précisant bien le nombre de bascules utilisées et un diagramme de transitions complet.
- Proposer un programme VHDL qui réalise cette nouvelle solution.
- On synthétise les deux versions du codeur dans un circuit programmable. Indiquer, pour chaque solution, quel circuit peut convenir parmi les PLDs 20 broches 16L8, 16R4, 16R8 et 16V8.
- Quels sont les avantages et inconvénients respectifs des deux solutions étudiées ?
- Proposer une solution pour le décodeur (deux entrées, une sortie).

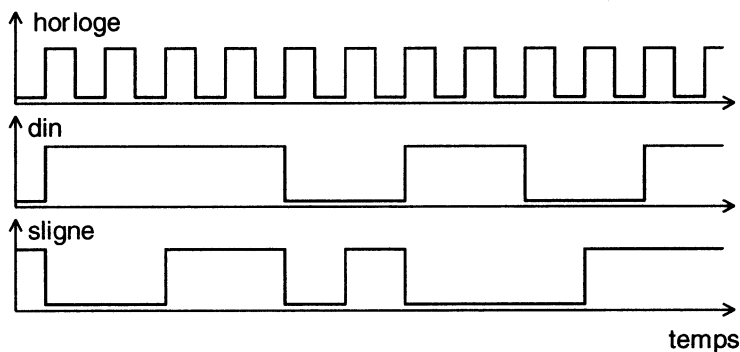
## 8. Exercice de synthèse : codeur CMI

Dans les transmissions numériques par infrarouge, télécommandes par exemple, on utilise souvent un code d'émission, dit code ligne tel que chaque bit émis est transmis sur *deux* périodes de l'horloge. On se propose ici de réaliser un codeur qui transforme des données binaires d'entrée, *din*, qui arrivent à raison d'un bit toutes les *deux* périodes d'horloge, en un code ligne, *sligne*, qui est construit de la façon suivante :

```

si din = '0' : sligne = '0' pendant une période d'horloge
               puis sligne = '1' pendant une période d'horloge.
si din = '1' : sligne = '0' pendant deux périodes d'horloge
               ou sligne = '1' pendant deux périodes d'horloge,
               en alternance.
  
```

L'alternance signifie que le niveau correspondant à un '1' logique pour *din* change d'une fois à l'autre, que les '1' successifs soient ou non séparés par des '0'. Le chronogramme ci-dessous donne un exemple de transmission :



Le codeur qui transforme *din* en *sligne* reçoit en entrée l'horloge et *din*. Il fournit en sortie *sligne*, retardée d'une période d'horloge, car le plus simple est de concevoir une machine de MOORE.

- À combien d'états internes correspond l'émission d'un bit ?
- Pourquoi les états correspondant à l'émission des codes pour des valeurs *din* = '1' successives ne peuvent-ils pas être toujours les mêmes ? (évident)
- Pourquoi les états correspondant à l'émission des codes pour *din* = '0' ne peuvent-ils pas être toujours les mêmes ? (question plus difficile)
- Dédurre de l'analyse précédente le nombre d'états que doit posséder le codeur.
- Proposer une ébauche de diagramme de transitions. On nommera les états par des noms, par exemple *pzero0* et *pzero1* pour l'émission d'un zéro, dans l'un des cas analysée au point c. À ce niveau on représentera les transitions importantes, mais pas forcément toutes les transitions possibles.
- Le codeur peut, en début d'émission, ne pas être synchronisé correctement. Compléter le diagramme précédent pour garantir qu'il se synchronise aussi vite qu'il peut le faire.

- Choisir un codage intelligent pour les états.
- Proposer une solution VHDL au problème.
- Proposer une solution pour le décodeur.

## 9. Analyse : comparaison de deux réponses à un même problème

Pour réaliser un codeur AMI (la connaissance de ce code n'est pas nécessaire à la compréhension du sujet), deux concepteurs différents proposent deux solutions, toutes deux justes, concrétisées par deux architectures (amidec et amidir) décrivant la même entité. Les résultats de simulation fonctionnelle des deux solutions sont identiques (voir figures en annexe).

Les programmes sources sont donnés en annexe.

Chacune de ces deux solutions est synthétisée et implémentée dans un circuit programmable AMD 16V8H-25 ( $t_{PD} = 25$  ns,  $t_{CO} = 12$  ns,  $F_{maxint} = 40$  MHz).

- Pour chaque solution indiquer la structure (pas les équations détaillées) du circuit généré. Préciser soigneusement les natures, combinatoires ou séquentielles, des différents blocs fonctionnels. Dédurre de chaque programme la dimension du registre d'état correspondant. Construire les diagrammes de transitions associés.
- On teste les circuits réalisés avec une horloge à 40 Mhz.
- Les résultats des deux tests sont fort différents, comme l'attestent les chronogrammes fournis en annexe. Interprétez quantitativement ces chronogrammes en vous appuyant sur les documents constructeur. Pour faciliter l'analyse on a placé des curseurs à des instants intéressants, et demandé l'affichage de l'écart temporel entre les curseurs.
- Conclure.

## 10. Annexe : programmes

Entité commune :

```
entity amicod is
    port(      hor, din : in bit ;
           plusout, moinsout : out bit );
end amicod ;
```

Architecture « amidec » :

```
architecture amidec of amicod is
    type ami is (mzero,pzero,moins,plus) ;
    signal etat : ami ;
    begin
        plusout <= '1' when etat = plus else '0' ;
        moinsout <= '1' when etat = moins else '0' ;
        encode : process
        begin
            wait until hor = '1' ;
            case etat is
                when mzero =>    if din = '1' then etat <= plus ;
                                end if ;
```

```

        when pzero =>    if din = '1' then etat <= moins ;
                        end if ;
        when moins =>    if din = '1' then etat <= plus ;
                        else etat <= mzero ;
                        end if ;
        when plus =>     if din = '1' then etat <= moins ;
                        else etat <= pzero ;
                        end if ;
        when others =>   etat <= mzero ;
    end case ;
end process encode ;
end amidec ;

```

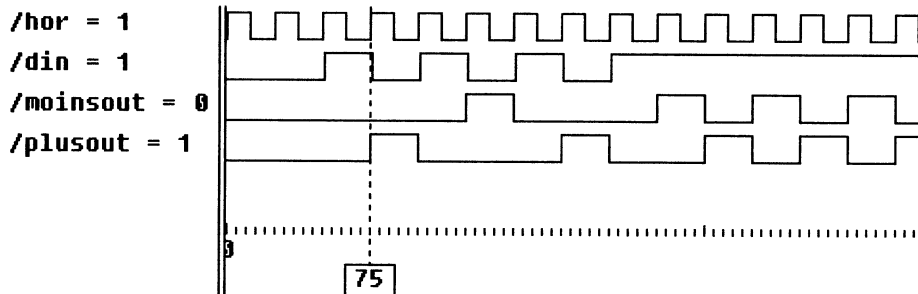
#### Architecture « amidir » :

```

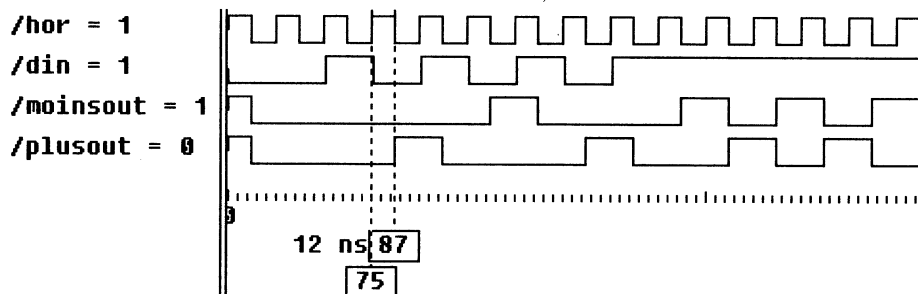
architecture amidir of amicode is
    subtype ami is bit_vector(2 downto 0) ;
    constant mzero : ami := "000" ;
    constant pzero : ami := "001" ;
    constant moins : ami := "010" ;
    constant plus  : ami := "100" ;
    signal etat : ami ;
begin
    plusout <= etat(2) ;
    moinsout <= etat(1) ;
    encode : process
    begin
        wait until hor = '1' ;
        case etat is
            when mzero =>    if din = '1' then etat <= plus ;
                            end if ;
            when pzero =>    if din = '1' then etat <= moins ;
                            end if ;
            when moins =>    if din = '1' then etat <= plus ;
                            else etat <= mzero ;
                            end if ;
            when plus =>     if din = '1' then etat <= moins ;
                            else etat <= pzero ;
                            end if ;
            when others =>   etat <= mzero ;
        end case ;
    end process encode ;
end amidir ;

```

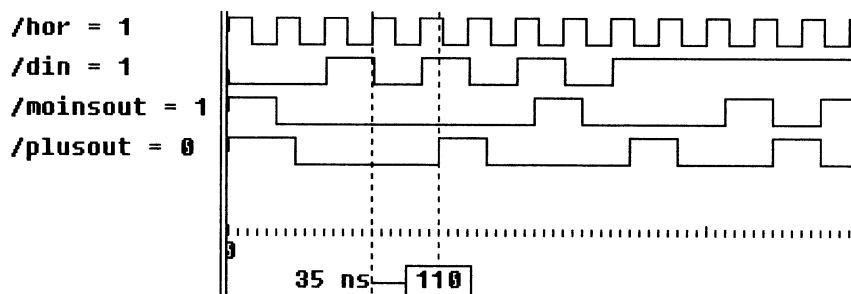
### 11. amidec ou amidir : simulation fonctionnelle du programme source



### 12. amidir : fonctionnement du circuit



### 13. amidec : fonctionnement du circuit



### 14. Étude d'un filtre numérique

Un filtre numérique récursif du premier ordre peut être modélisé par le programme ci-dessous :

```

entity passe_bas is
  port(hor : in bit ;
        raz : in bit ;
        din : in real ;
        dout : out real := 0.0 ) ;

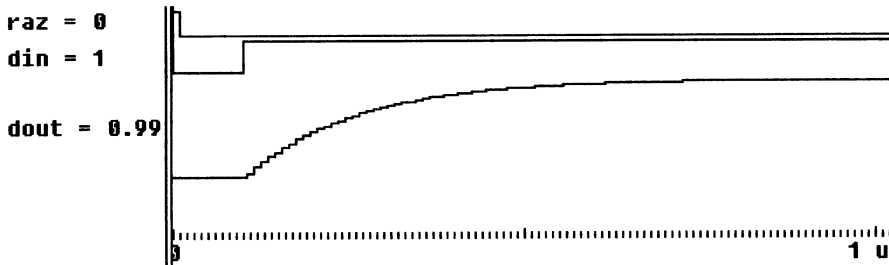
```

```

end passe_bas ;
architecture comporte of passe_bas is
    signal dinret, sort, sortret : real := 0.0 ;
begin
    sort <= (30.0 * sortret + din + dinret)/32.0 ;
    retards : process
    begin
        wait until hor = '1' ;
        if raz = '1' then
            dinret <= 0.0 ;
            sortret <= 0.0 ;
            dout <= 0.0 ;
        else
            dinret <= din ;
            sortret <= sort ;
            dout <= sort ;
        end if ;
    end process retards ;
end comporte ;

```

Une simulation fonctionnelle fournit la réponse à un échelon de ce filtre :



Dans les courbes précédentes les échelles verticales des signaux d'entrée et de sortie ne sont pas identiques, le gain statique du filtre est égal à 1 (on ne demande pas de le démontrer).

- Ce filtre n'est pas synthétisable dans un circuit programmable de complexité raisonnable. Pourquoi ?
- Une première version synthétisable de ce filtre est donnée par le programme :

```

entity passe_bas is
    port(hor : in bit ;
          raz : in bit ;
          din : in integer range 0 to 255 ;
          dout : out integer range 0 to 255 := 0 ) ;
end passe_bas ;

architecture comporte of passe_bas is
    signal dinret, sort, sortret : integer range 0 to 255 := 0 ;

```



```

begin
sort <= (30 * sortret + din + dinret)/32 ;
retards : process
begin
    wait until hor = '1' ;
    if raz = '1' then
        dinret <= 0 ;
        sortret <= 0 ;
        dout <= 0 ;
    else
        dinret <= din ;
        sortret <= sort ;
        dout <= sort ;
    end if ;
end process retards ;
end comporte ;

```

Combien de bascules doit contenir, au minimum, le circuit cible ?

On applique un échelon d'amplitude 255 à l'entrée du filtre, quelle devrait être, idéalement, la valeur finale de la sortie ?

- La valeur finale atteinte par la version précédente du filtre est 240. Quelle est l'origine de ce dysfonctionnement ? (pour trouver la réponse on pourra, par exemple, se poser la question de la réponse du filtre à un échelon d'amplitude inférieure à 16.)
- Une version corrigée du programme précédent correspond à l'architecture fournie ci-dessous :

```

architecture comporte2 of passe_bas is
    signal dinret : integer range 0 to 255 := 0 ;
    signal sort, sortret : integer range 0 to 8191 := 0 ;
begin
sort <= (15 * sortret)/16 + din + dinret ;
retards : process
begin
    wait until hor = '1' ;
    if raz = '1' then
        dinret <= 0 ;
        sortret <= 0 ;
        dout <= 0 ;
    else
        dinret <= din ;
        sortret <= sort ;
        dout <= (sort + 16)/32 ;
    end if ;
end process retards ;
end comporte2 ;

```

Quelle est l'amélioration ?

- Quel est le coût de cette amélioration en terme de complexité du circuit cible du processus de synthèse ? (justifier cette augmentation de complexité.)

- Une autre version correcte du filtre est celle ci-dessous :

```
architecture comporte3 of passe_bas is
    signal sort, retard : integer range 0 to 8191 := 0 ;
begin
    sort <= retard + din ;
    retards : process
    begin
        wait until hor = '1' ;
        if raz = '1' then
            retard <= 0 ;
            dout <= 0 ;
        else
            retard <= (15*sort)/16 + din ;
            dout <= (sort + 16)/32 ;
        end if ;
    end process retards ;
end comporte3 ;
```

Montrer que la fonction réalisée est la même. Qu'a-t-on gagné ?

- À quelle opération simple correspond une division par 16 ou par 32 ?
- Toutes les versions précédentes utilisent une multiplication, opération complexe. Certains outils de synthèse n'acceptent pas cette opération dans le cas général. Comment pourrait-on contourner cette difficulté ? (On proposera une modification au dernier programme.)
- Le filtre étudié précédemment a une constante de temps liée de façon rigide à la période de son horloge. Comment pourrait-on créer un filtre du premier ordre « universel », dont la constante de temps soit paramétrable ? Quel problème cela pose-t-il en synthèse ?

# Bibliographie

## Méthodes de synthèse et VHDL

- R. AIRIAU, J.-M. BERGÉ, V. OLIVE, J. ROUILLARD — *VHDL du langage à la modélisation*, Presses polytechniques et universitaires romandes, 1990.
- J. WEBER, M. MEAUDRE — *Circuits numériques et synthèse logique, un outil : VHDL*, Masson, 1995.
- P. LARCHER — *VHDL : introduction à la synthèse logique*, Eyrolles, 2000.
- Z. NAVABI — *VHDL : Analysis and Modeling of Digital Systems*, McGraw Hill, 1993.
- K. SKAHILL — *VHDL for programmable logic*, Addison-Wesley, 1996
- R. LIPSETT, C. SCHAEFER, C. USSERY — *VHDL : hardware description and design*, Kluwer Academic, 1989.
- J. ARMSTRONG — *Chip Level Modelling in VHDL*, Prentice Hall, 1988.
- Introduction to VHDL*, Mentor Graphics, 1992.
- VHDL Reference Manual*, Mentor Graphics, 1994.
- AutoLogic VHDL Reference Manual*, Mentor Graphics, 1994.
- IEEE Standard VHDL Language Reference Manual, Std 1076-1993*, IEEE, 1993.
- IEEE Standard 1076 VHDL Tutorial*, CLSI, 1989.
- Librairies des logiciels SYNARIO (DATA IO), V-SYSTEM (Model Tech), WARP (CYPRESS), FPGA EXPRESS (SYNOPSIS).
- J.-M. BERNARD, *Conception structurée des systèmes logiques*, Eyrolles, 1987.
- PAL device Handbook*, Advanced Micro Devices, 1988 [référence ancienne qui contient une excellente introduction à la synthèse logique].
- F.J. HILL, G.R. PETERSON — *Computer aided logical design with emphasis on VLSI*, John Wiley & sons, 1993.
- C. MEAD, L. CONWAY — *Introduction to VLSI systems*, Addison-Wesley, 1980.

## Circuits et opérateurs logiques

Ch. TAVERNIER — *Circuits logiques programmables*, Dunod, 1996

D.A. HODGES, H.G. JACKSON — *Analysis and design of digital integrated circuits*, McGraw-Hill, 1988.

J. MILLMAN, A. GRABEL — *Microelectronics*, McGraw Hill, 1988 [existe en traduction française chez le même éditeur].

*FPGA data book and design guide*, ACTEL.

*Data Book*, ALTERA.

*Programmable Logic*, Cypress.

*Data Book, Handbook*, Lattice.

*FPGA Applications Handbook*, Texas Instrument.

*The Programmable Logic Data Book*, Xilinx.

# Index

## A

- access 35
- affectation 53
- agrégat 45
- alias 40
- anti-fusible 199
- architecture 27
- array 34
- assert 148
- association 60
- asynchrone 79
- attribut 46
  - prédéfini 48
  - utilisateur 51
- attribute 51

## B

- Backus et Naur 15
- banc de test 169
  - en boucle fermée 176
- bascules, types 195
- behavioral 29
- bit 32
- bit\_vector 34
- bloc 62
  - externe, interne 120
  - implicite, generate 122
- block 63
- BNF 15

- boolean 32
- boucle 72
  - espace et temps 99
  - fermée (simulation) 176
  - generate 122
  - non synthétisable 99
- brochage (d'un composant) 51
- burried flip flops 202
- bus 38, 132
  - signal gardé 136

## C

- case, décodeur 78
- case ... when 71
- causalité 57
- cellules SRAM 199
- chaîne
  - binaire 45
  - caractères 45
- classe 31
  - file 143
- clock skew 103
- codage des états 87
- code passif 104, 149
- commandes asynchrones 82
- commentaire 9
- composant 59
  - configuration 125
- concurrency 52
- conditionnelle (affectation) 54

- configuration 124
  - composant 125
  - de test 173
  - port/generic map 125
- conflits 132
- constant 37
- constante 36
  - littérale 44
- construction hiérarchique 12
- conversion de type 46
  - instanciation 122
- CPLD 189
- cycle delta 56
- D**
- data flow 29
- décodeur 75
  - algorithme séquentiel 77
- défaut (valeur par) 38
- description
  - comportementale 29
  - flot de données 29
  - structurelle 29
- driver 37, 67
  - et signaux résolus 132
- E**
- edge 80, 81
- EEPROM 188
- enregistrements 35
- entité 24
- entity 24
- entrée standard 116
- énuméré (type) 32
- EPROM 188
- erreurs 148
- étiquette 53
- événement 55, 64, 67
- exit 74
- expression 41
  - qualifiée 46
  - statique 106
- F**
- falling\_edge 139
- feu tricolore
  - fonctionnel 18
  - synthétisable 21
- fichier
  - déclaration, ouverture et modes 143
  - paquetage textio 116

- fichier source 113
- file 36, 143
- file\_open\_kind 143
- finite state machine 83
- fitter 202
  - et rétroannotation 157
  - son rôle 212
- FLASH 188
- flot de données
  - décodeur 77
- fonction 104
  - de résolution 132
  - syntaxe 105
- for 74
  - configuration 125
- forme d'onde 53
- formel (paramètre) 60
- FPGA 189
- fréquence maximum, modèle 207
- fusibles 195

**G**

- generate 122
- generic map 60
  - configuration 125
- génériques 27, 120
- guarded 38, 54

**H**

- hiérarchie 119
  - schémas algorithmiques 123
- hold, modélisation 154
- horloge 81
  - et synthèse 131
  - std\_logic 139

**I**

- IEEE
  - fonctions de conversion 138
  - librairie 117
  - normes 6
  - opérateurs 140
  - paquetage ieee.std\_logic\_1164 137
  - paquetages 118
  - VITAL 159
- if ... then 71
- impure (fonction) 150
- indexed name 44
- inertial 153

input 116, 147  
instanciation 59  
integer 33

## J

JEDEC 196  
JTAG 203

## K

kernel 58

## L

latch 80  
    cachés 92  
bibliothèque IEEE 117  
liste  
    de sensibilité 64, 68  
    d'associations 60  
\_logic 136  
loop 74  
LPM 208

## M

machines à nombre fini d'états 83  
    codage 87  
    Mealy 88  
    Moore 86  
macrocellules 201  
mémoire 72  
mode 26  
modules primaires et secondaires 113  
MOS (grilles flottantes) 197  
multiplexeur  
    architecture 28  
    entité 25  
    principe 192

## N

natural 34, 116  
next 74  
nom 43  
    attribut 44  
now (fonction) 150  
noyau (du simulateur) 58  
null  
    forme d'onde 54, 151  
    instruction 72  
numeric\_std/bit (IEEE) 139

## O

objet 36  
one hot 88  
opérande 43  
opérateur 41  
    algorithme séquentiel 77  
    combinatoire 75  
    bibliothèque IEEE 140  
    séquentiel 78  
    surcharge 111  
    synchrone, asynchrone 79  
output 116, 147  
overloading 110

## P

package 114  
paquetage 114  
parallélisme 57  
paramètres génériques 60  
parité  
    boucle 73  
    faux 96  
    paquetage 114  
    sous-programmes 109  
pièges 91  
    faux trois états 129  
pilote 37, 67  
pipe line 102  
placement 202  
PLD 189  
pointeur et fichiers 148  
polarité programmable 194  
port map 60  
    configuration 125  
ports, *Voir* entité  
positive 35, 116  
PREP 209  
procédure 104  
    syntaxe 105  
processus 64  
    combinatoire 95  
programmable  
    *in situ* 198  
    logic array 190  
projected waveform 52, 69  
    et temps simulateur 151  
PROM 188  
pull up/down 136

**R**

RAM 189  
 read 117, 144  
 read\_mode 117  
 readline 117  
 real 34  
 record 35  
 réel (paramètre) 60  
 register 38  
   signal gardé 136  
   transfert language 29  
 registre  
   à décalage 70, 74  
   d'état 85  
 reject 153  
 report 150  
 réseau logique programmable 190  
 resolved (fonction de résolution) 138  
 retard, modélisation 151  
 rétroannotation 10  
   modélisation 156  
 return 106  
 réveil (processus) 64, 67  
 rising\_edge 82, 139  
 routage 202  
 RTL 29

**S**

schéma d'itération (boucles) 74  
 selected name 43  
 sélection (instruction séquentielle) 71  
 sélective (affectation) 54  
 sémaphore 52  
 sens caché et synthèse 129  
 séquentiel (processus) 66  
 set up, modélisation 154  
 severity level 148  
 signal 37  
   affectation  
     concurrente 53  
     séquentielle 65, 69  
   concurrence 52  
   et variable 96  
   gardé 38  
   résolu 136  
   valeur par défaut 38  
 signature d'un sous-programme 111  
 signaux d'interface 25

simulation  
   en boucle fermée 176  
   et synthèse 128  
   fonctionnelle 8  
   outils 141  
   rétroannotée 10  
 slice name 44  
 sommeil (processus) 64, 67  
 sortie standard 116  
 sous-programme 104  
   parité 109  
   syntaxe 106  
 spécification de contexte 113  
 standard (paquetage) 116  
 standard delay format (VITAL) 160  
   exemple 163  
 std (librairie) 116  
 std\_ulogic 136  
 string 35  
 surcharge 110  
   opérateurs 111  
 synchrone 79  
 synthèse 6, 8  
   boucles 99  
   complexité 101  
   et simulation 128  
   feu 21  
   horloges 103, 131  
   pièges 91  
   sens caché 129  
   signaux et variables 96  
   trois états 130

**T**

table de vérité 75  
 tableau 34  
 test (instruction séquentielle) 71  
 test bench 142  
   exemples 165  
 textio  
   utilisation 144  
   paquetage 116  
 time 33  
 top down design 119  
 tranche 44  
 transaction 52  
   signaux résolus 132  
 transport 153  
 trois états (sorties) 135  
 trois zéros (machine d'états) 85



type 30  
  conversion 46  
  entier 33  
  énuméré 32  
  file 143  
  flottant 33  
  physique 33  
  pointeur 35  
  précision 46  
  prédéfini 32  
  scalaire 32  
  structuré 34  
  syntaxe 31

**U**

type, std 136  
unaffected 54  
unité de conception 112  
use 113

**V**

variable 39  
  affectation 65, 68  
  durée de vie 40

  et bascule 97  
  et signal 96  
  locale 108  
vecteur 34  
  et nombre 139  
  std\_logic 138  
VHDL 6  
VITAL 159

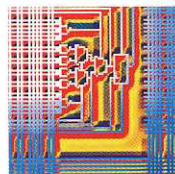
**W**

wait 64, 67  
  et synthèse 132  
  procédures et fonction 105  
waveform 151  
when ... else 53  
while 74  
with ... select 53  
work (librairie) 116  
write 144

**Z**

zone déclarative 27  
  sous programmes, définitions 104

Jacques Weber  
Maurice Meaudre



# LE LANGAGE VHDL

## Cours et exercices

2<sup>e</sup> édition

La seconde édition de ce manuel sur l'utilisation du langage VHDL s'adresse aux électroniciens et aux informaticiens, ainsi qu'aux étudiants de ces disciplines qui souhaitent maîtriser la modélisation et la réalisation des circuits numériques.

Le langage VHDL est devenu un véritable outil de conception des circuits numériques. Il permet d'autre part de concevoir et de vérifier un système électronique complexe. Cet ouvrage expose en trois parties les étapes qui conduisent de la simulation au circuit opérationnel.

La première partie aborde rapidement le contexte de la conception assistée par ordinateur des circuits numériques. La deuxième développe la structure et les possibilités du langage VHDL, tant en synthèse qu'en vérification. De nombreux exemples permettent au lecteur de valider sa compréhension.

La dernière partie est une présentation synthétique des circuits programmables. Des énoncés d'exercices sont proposés à la fin du cours.

En complément à l'utilisation de ce manuel, des liens vers les fournisseurs de logiciels de simulations, des fichiers de programmes, les solutions des exercices et des énoncés d'exercices supplémentaires sont disponibles et téléchargeables sur le site Web des auteurs.

**WWW** fichiers de simulation et exercices sur le Web



ISBN 2 10 004755 8

<http://www.dunod.com>

JACQUES WEBER  
est maître de conférences  
à l'IUT de Cachan.

MAURICE MEAUDRE  
est chef de travaux  
d'Ensam à l'IUT de  
Cachan.

